Affinity: A Concurrent Programming System for Multicomputers

Craig S. Steele

Computer Science Department
California Institute of Technology

Caltech-CS-TR-92-08

Affinity:
A Concurrent Programming System for Multicomputers

Thesis by
Craig S. Steele

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1992

(Submitted May 27, 1992)

Caltech-CS-TR-92-08

Acknowledgments

Not Art and Science serve, alone;
Patience must in the work be shown.
A quiet spirit plods and plods at length;
Nothing but time can give the brew its strength.

Goethe, *Faust I*

I would particularly like to thank my research advisor, Chuck Seitz, for his patience, impatience, and insistence that simplicity is the essence of good research. The other members of my examining committee, Yaser Abu-Mostafa, Mani Chandy, Doug Rees, and Steve Taylor, deserve gratitude for volunteering to plow through the muddy fields of another person's mind.

I owe incalculable debt to past and present students in the group who have persisted in attempting to penetrate the equally baffling depths of concurrent systems design and my exposition: Wen-King Su, Don Speck, Jakov Seizović, Lena Peterson, Mike Pertel, John Ngai, Sven Mattisson, Charles Flaig, Nanette Boden, and particularly Bill Athas, for his services in his roles as officemate, *Amos Throop* crewmember, and consulting adventurer. Without the help of Arlene DesJardins, we'd all be ... well, without. I am indebted to Chris Lee for consorting with daemons on my behalf. I owe special thanks to Laura Jones for her proofreading and many helpful suggestions.

I'd like to thank my parents and aunt for their support and love, and my brother for tolerating a near-permanent database bug while I wrote this thesis. My sons Benjamin and Quinton are constant reminders of the innate power of inquiry; their joy in discovery is a continuing inspiration. The small children described in this section were sponsored in large part by my wife, Vicky, who is also the monitoring agency.

The research described in this thesis was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) Submicron Systems Architecture Project.

Abstract

Affinity is an experiment to explore a simple, convenient, and expressive programming model that provides adequate power for complex programming tasks while setting few constraints on potential concurrency. Although the programmer is required to formulate a computational problem explicitly into medium-sized pieces of data and code, most of the additional functions necessary for concurrent execution are implicit. The execution of the light-weight, reactive processes, called *actions*, implicitly induces atomicity and consistency of data modifications. The programmer accesses shared data structures in a shared-memory fashion, but without the need for explicit locking to manage the problems of concurrent access and mutual exclusion. Program control flow is distributed and implicit.

The name given to the programming model, Affinity, has a definition, "causal connection or relationship," that is fitting to the way programs are structured and scheduled.

Affinity consistency and coherence properties provide a tractable discipline for the dangerous power of a concurrent, shared-memory programming style. Existing programming complexity-management techniques such as object-oriented languages can be used in this multicomputer environment. Affinity programs can compute consistent and correct results despite staleness of data, and asynchrony and nondeterminism in execution of code. Program correctness is invariant under replication, or *cloning*, of actions. This aspect of the model yields a simple and robust mechanism for fault-tolerance.

The practicality of the Affinity programming model has been demonstrated by an implementation on a second-generation multicomputer, the Ametek S/2010. The implementation is distributed, scalable, and relatively insensitive to network latency. Affinity has demonstrated reasonable efficiency and performance for computations with tens of processing nodes, hundreds of actions, and thousands of shared data structures.

Contents

List of Figures

Chapter 1

Introduction

Multicomputers are concurrent, scalable, distributed-memory computing systems composed of many computational nodes linked together by a high-performance message-passing communications system [5]. This computer architecture offers otherwise unrealizable performance and cost-effectiveness for many computational problems. Multicomputer hardware performance has improved due to advances both in the generic processor and memory technologies, and in more specific insights into message-passing subsystems. Second-generation multicomputers [43, 42, 44] improved communications performance by one or two orders of magnitude relative to earlier machines by exploiting new routing technologies [19, 20, 36, 39]. This enhanced resource, in combination with more sophisticated processor architectures, has provided an opportunity to experiment with programming styles that are not based on explicit message passing, and are less closely coupled to the underlying hardware realization.

1.0.1 The Affinity Computational Model

Affinity is an experiment to explore a simple, convenient, and expressive programming model that provides adequate power for complex programming tasks while setting few constraints on potential concurrency. Although the programmer is required to formulate a computational problem explicitly into medium-sized pieces of data and code, most of the additional functions necessary for concurrent execution are implicit. The execution of the light-weight, reactive processes, called *actions*, implicitly induces atomicity and consistency of data modifications. The programmer accesses shared data structures in a shared-memory fashion, but without the need for explicit locking to manage the problems of concurrent access and mutual exclusion. Program control flow is distributed and implicit.

The name given to the programming model, Affinity, has a definition, "causal connection or relationship," that is fitting to the way programs are structured and scheduled.

1.0.2 Principal Features of the Model

Affinity consistency and coherence properties provide a tractable discipline for the dangerous power of a concurrent, shared-memory programming style. Existing programming complexity-management techniques such as object-oriented languages can be used

in this multicomputer environment. Affinity programs can compute consistent and correct results despite staleness of data, and asynchrony and nondeterminism in execution of code. Program correctness is invariant under replication, or *cloning*, of actions. This aspect of the model yields a simple and robust mechanism for fault-tolerance.

1.0.3 Experimental Implementation

The practicality of the Affinity programming model has been demonstrated by an implementation on a second-generation multicomputer, the Ametek S/2010 [43, 42, 44]. The implementation is distributed, scalable, and relatively insensitive to network latency. Affinity has demonstrated reasonable efficiency and performance for computations with tens of processing nodes, hundreds of actions, and thousands of shared data structures.

1.1 Programming Models

1.1.1 Message-Passing Models

Multicomputers are commonly programmed in a multiple-process style using explicit message-passing communications functions. The process model is familiar, and the programming is done with an augmented standard language [41, 44, 45]. This has proven to be a powerful and efficient approach, but it is perceived as conceptually difficult by some, and tediously detailed by many.

Programmers accustomed to conventional sequential computing find aspects of the message-passing style troublesome:

Encapsulated Data - Data is contained in and owned by individual processes. Either the computation must be *supply-driven* (with processes sending unsolicited messages to predictable recipients), or data-access services must be incorporated into the code.

Process Asynchrony - If data communication requires closely-coupled process interaction, program structure can be complicated if processes run at different speeds. This is further inducement for a supply-driven programming style and substantial system message buffering.

Machine-Dependence - While it is, in principle, an orthogonal issue, some message-passing systems require the user to embed hardware-platform-specific information, such as processor-node numberings, into the code. Direct management of process and data location is annoying and nonportable.

Writing explicitly concurrent asynchronous process code can be quite complex in a message-passing model, particularly for large-process monoliths which must provide many access modes to contained and owned data.

Many programmers respond to these challenges by embracing programming idioms that restrict communications to simple and regular logical patterns (predefined grids, etc.) and synchronous alternation of computation and data exchange phases [23, 38]. For particularly regular problems, this path can produce simple and efficient code. For

applications not ideally suited to such Procrustean treatment, the computation's complexity is not eliminated but rather transmuted into recurrent preoccupations with load balance and data distribution. Unfortunately, such problems tend to be exacerbated by increasing problem scale and complexity.

1.1.2 Shared-Memory Models

Shared-memory multiprocessors have a global address space and elaborate supporting memory hardware to eliminate the problems of data access, distribution, locality, and staleness. The characteristic logical error in programming such machines is failing to adequately restrict (interlock) access to shared data; in effect, data sharing is too easy. Both software and hardware have inherent difficulties in scaling beyond modest numbers of concurrent components, typically evidenced as "hot-spot" variables and escalatory memory-switch complexity, respectively. Perhaps because there is an incremental path from purely sequential uniprocessor code to concurrent refinements, shared-memory machines are commonly but incorrectly regarded as easier to program than distributed-memory machines. The prospect of extending the techniques that have been developed for modestly concurrent pipelined vector processors seems attractive to many workers.

1.1.3 Distributed Shared-Memory Models

Distributed shared memory (DSM) [37] is another effort to build upon an existing model. It is not particularly difficult to implement some form of a shared virtual-address space across distributed hardware, e.g., a network of workstations or a multicomputer [32, 21]. However, such a distributed shared memory has dramatically reduced performance compared to a physically-shared memory, and will perform poorly if programmed in the same manner. In and of itself, DSM is simply a raw mechanism that can be used to support message-passing or file analogs. Absent some semantically significant policy, DSM extends the tyranny of the pointer to a larger stage, and allows clever programmers to write complicated programs.

Nevertheless, decades of development in support of memory-based programming models have produced powerful and efficient languages, processors to execute them, and legions of adepts. The Affinity programming model strives to tame the power and abate the potentially insatiable demands of the DSM approach.

1.1.4 Concurrent Languages

Many researchers have crafted explicitly concurrent languages. One school of particular interest is the Actor model [1]. As realized in the Cantor experiment [4, 5, 11], Actors provided a practical understanding of the issues of implementation and expressivity in fine-grain programming. Actor computations depend upon the concept of *reactivity* [53, 45], the notion that programs run in response to an event such as message arrival.

Whereas the Actor paradigm emphasizes the mutability of behavior, another approach has a more declarative style. The UNITY concurrent programming model [14] has considerable overlap with the Affinity model, particularly in its foundation upon an assignment model and embrace of nondeterminacy. More recent Caltech work, such as PCN, has emphasized implicit synchronization by data access [15, 22].

The popularity of object-orientation is such that it is common to hear loose conjectures that recoding a program in an O-O language will parallelize any algorithm. Nevertheless, there are interesting experiments in extending O-O programming languages to embrace some semantic concurrency, e.g., by incorporating the notion of futures into C++ [48, 46].

1.2 Yet Another Programming Model?

> Concurrent programming is not an easy subject in which to achieve mastery. It contains many subtleties and traps for the unwary over and above all the usual complexities of 'ordinary' sequential programming [6].

What can justify introducing yet another new model for concurrent programming? First, opportunism. The field appears to remain open, since programmers remain generally unweaned from sequential, single-process programming. Concurrent programming retains an arcane mystique; in some circles, nondeterminacy is regarded as a synonym for error. In presenting Affinity, we hopefully suggest that programmers can accept some conceptual novelties regarding relaxed data coherence in exchange for relief from less abstract chores such as hardware resource management, message operations, lock maintenance, and large-scale control flow. The Affinity atomic action model pares down the reactive, concurrent process model to a more comprehensible logical unit. While the leading quotation was selected to emphasize the intimidating reputation of concurrent programming, it does contain an essential truth: a concurrent programming style that simply adds to a sequential model without providing some offsetting simplification will be relatively harder.

Second, anticipated necessity. If we project the characteristics of a grandly scaled multicomputer, we can make two conclusions about the hardware, one physical, one economic. First, it will be loosely synchronized, with significant and probably nonuniform data-access latencies. Second, it will typically be partially failed and failing. A programming model that cannot tolerate these two imperfections will be relatively costly even where heroic hardware measures are employed, and unreliable where they are not. The Affinity model has some intrinsic advantage in both respects.

Third, skepticism. Efforts to discover concurrency automatically from existing sequential programming languages are somewhat successful, since some useful idioms have been identified by programmers and compiler writers, but rarely yield more than ten-fold concurrency. Larger-scale concurrency still seems to require either a different programming model, i.e., message-passing, or a supplementary annotation language detailing the potential parallelism of either the operation or the data organization. It would seem that the challenge of mechanically inferring where and when data staleness might be tolerable would be even more daunting. Affinity requires that the programmer partition the data and program code to expose the problem's concurrency, but substantially decouples these choices from considerations of fit or efficiency on the underlying hardware. The mapping to the hardware should be at most an afterthought to the logical structure of the program, not *vice versa*.

1.3 Affinity in Short

The Affinity concurrent-programming environment is an evolution of the reactive programming model for medium-grain programming on second-generation multicomputers. Virtual-memory hardware is used to support a distributed, shared-memory programming system based on low-context processes called *actions*. The programmer accesses shared data structures, called *blocks*, much as in a shared-memory machine, but without the need for explicit locking for concurrency control. The execution cycle of actions implicitly induces atomicity and consistency of effect in modifying computational state. Mutually-exclusive update of all modified data is implemented by the definition of atomic effect, which guarantees that changes are effected completely or not at all. The rule is applied to the set of data blocks actually accessed during execution, the *write set*. Data in the write set are known to be the current versions. Computations that are tolerant of data staleness may improve concurrency by relaxed coherency requirements for read data; the triggering mechanism for scheduling makes this class of problems surprisingly large.

1.3.1 Distributed and Nondeterministic Scheduling

Scheduling of actions for execution is distributed and nondeterministic. Actions may set *triggers* on data blocks. When such a data block is modified, all actions with set triggers will be scheduled for subsequent execution. This externalization of the computation's control flow allows programs to be written as small code fragments that establish particular relations between input and output data objects. In common with UNITY, at termination of the computation, the desired relations are established on the final results.

Scheduling actions in reaction to data-modification events directly simplifies programming in two ways: by removing explicit control structures and event-handling code, and through decentralization, allowing the coding unit to be significantly reduced in size. (The simplified access to shared data also helps in this goal. The task of decoding a single input stream is tedious and has given rise to programmer demand for various typed-message schemes.) Indirectly, reactivity allows a tolerance for data staleness. Suboptimal or "inconsistent" intermediate results can be generated without harm if it is known that they will be discarded later when a better result will be computed.

1.3.2 Optimistic Execution

Efficiency, or more precisely, scalable efficiency, is the *sine qua non* of concurrent computation, but it is the aggregate efficiency that determines the total runtime of a computation. Affinity is designed under optimistic assumptions that place few restrictions on concurrency for a properly designed program, but may perform considerable redundant and unusable computation in some situations. It is presumed that for an appropriately programmed computation, such wasted processor cycles result from an inherent sequentiality in part of a problem, and could not be effectively used in any case.

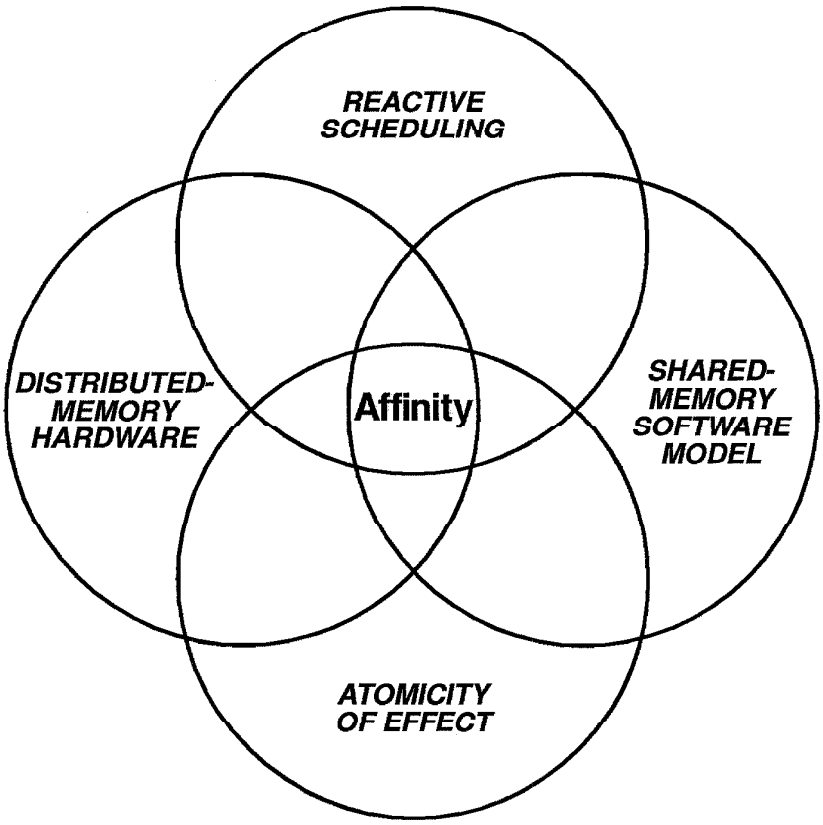

Figure 1.1: Basic ideas underlying Affinity. The combination of multicomputer distributed-memory hardware and a shared-memory programming model motivate the definition of Affinity *data blocks*. The combination of reactive scheduling and atomicity of effect motivate the definition of Affinity *actions*.

1.4 Relation to Other Work

Affinity has emerged from a rich soup of concurrent-programming research at Caltech that has extended over more than a decade [44]. The collective experience with Caltech-developed message-passing software (Cosmic Kernel (CK), Cosmic Environment (CE), Reactive Kernel (RK), and Reactive C) [40, 41, 44, 45, 53]), and extensions of the object-oriented language Simula [29], form the foundation and milieu for the Affinity experiment. The fine-grain Actors of the Cantor language [4, 5, 11] are obvious precursors to Affinity's medium-grain actions.

A complementary influence to the reactive programming model is UNITY [14], which

adopts an expressly nondeterministic scheduling policy. The UNITY programming notation is not specific to any particular implementation. This generality has advantages. UNITY's methods of reasoning can be applied to Affinity programs; conversely, UNITY programs can be straightforwardly expressed as Affinity programs. However, to actually produce a practical implementation of UNITY on a concurrent system requires some additional mechanisms. Affinity does provide them, perhaps the most obvious being the trigger mechanism for scheduling. One significant conceptual difference is that UNITY programs have a static structure, which aids analysis; Affinity program structure is built dynamically and can mutate. The peculiarities of Affinity atomicity and the variability of its data-coherence rules are rather too specialized to fit into UNITY's several architectural models. Despite these differences, the motivation and character of the two models are congruent.

The name of the trigger mechanism was inspired by a provocative discussion by Kotov [27], though the actual mechanism is unrelated.

1.4.1 Distributed Shared Memory

The field of distributed shared memory is increasingly popular [37, 21, 32]. As argued above, DSM in itself just gives programmers more rope with which to hang themselves; some restrictive semantic or linguistic model is required to give some structure and simplicity to this amorphous conceit. DSM models operate at a level too low for either simplicity of program structure or general-purpose efficiency. There is a poor match between the logical grain of the program, the variable, and the unit of communication and coherence, the page (several KB in size for most implementations). More sophisticated efforts allow a finer-grain structure to be defined within pages, and allow the user to implement the requisite operations such as merging and mutual exclusion [8, 13]. Languages such as Orca [7], which are based on a higher-level object-oriented model, seem to be much better suited to user-level programming.

1.4.2 Relaxed Data Coherence

The Affinity data block distribution problem is described in terms of a software-maintained cache-coherence mechanism. Efforts to develop a scalable hardware mechanism for cache coherence have significant similarities, particularly directory-based cache coherence schemes such as Dash [31] which incorporates a form of relaxed data coherence called "release consistency." It can be argued that there is a conceptual convergence between sophisticated approaches to cache-coherent multiprocessors and Affinity's emulation of a shared-memory model.

1.4.3 Distributed and Concurrent Databases

Database systems have long had to contend with asynchronous-access issues, both from planned concurrency and unplanned system failures. Although inspired by reactive-programming models and UNITY, Affinity action atomicity is similar to database-transaction models. Direct approaches that extend and elaborate the UNIX process model by adding explicit concurrency control and data-distribution mechanisms to conventional languages can become baroque. The number of legitimate combinations of

these language directives can become confusingly large. Simply melding a basic implementation of transactions with a distributed environment (or even a DSM environment [49]) produces unrewarding complexity.

The ObjectStore object-oriented database management system [28] is an innovative effort that shares many features and implementation mechanisms with Affinity. In common with Affinity, it uses virtual-memory hardware facilities to provide an object-based shared memory and to record object access. The standard ObjectStore coherence mechanism is the usual DSM write-invalidate scheme [37], but users can implement other rules on an application-specific basis. Since ObjectStore uses a conventional process model, the beginning and end of atomic transactions must be explicitly specified in the user code. ObjectStore surpasses Affinity in several aspects central to its database mission, e.g., supporting persistent objects and logging transactions to disk. However, ObjectStore is not intended as a concurrent programming language, and lacks the novel implicit scheduling and coherence mechanisms of Affinity.

1.5 Overview of Subsequent Chapters

In Chapter 2, we define the basic conceptual elements of Affinity programs. Two distinct notations for actions and data blocks are introduced for exposition. One, a graphic representation, is used in subsequent chapters. The other, based on an atomic assignment formula, is used only to explain Affinity coherence properties. Neither notation is the actual programming language used to implement the model.

Chapter 3 introduces that language, a C++ subset, in the context of a series of tutorial examples. The first short programs focus on the usage details of this implementation. Subsequent examples illustrate how the implicit consistency and coherence rules are used to design more interesting programs.

Chapter 4 is a moderately detailed description of the S/2010 implementation of Affinity. The implementation is relatively simple, partly because of its specialization, but mainly because it was possible to use the same mechanisms, such as data-block transport and action atomicity, for both user- and kernel-level functions. We take this as evidence of a certain rightness of the model's abstractions. Chapter 5 presents measurements of the performance and behavior of this implementation.

In Chapter 6, we conclude with a review of the work done and a discussion of further possibilities.

Chapter 2

Computational Model

The basic conceptual elements of Affinity programs are defined in this section; the next chapter will reintroduce them in the context of example programs.

Actions, the Affinity process analog, are the agents of execution of the programmer's code. A computation is performed by an aggregation of actions.

Data blocks are structures composed of one or more variables. Collectively, they contain the computational state.

Triggers specify an association between data blocks and actions. Triggers provide a practical action-scheduling mechanism by ensuring that changes to data blocks induce subsequent execution of associated actions.

The expositional prose is augmented by a somewhat more formal definition of the model in terms of atomic assignment. A graphic notation depicting the three basic elements is also introduced in this chapter; it will be used extensively to describe the example programs of the next chapter. The atomic-assignment and graphic notations are presented for expository purposes; neither is the actual language of the code examples, which is a subset of C++.

2.1 Actions

Affinity actions are the agents of program code execution, analogous to processes in other models. Actions operate on the content of data blocks and have minimal intrinsic state (see Section 2.10.1). Specifically, the state of program variables is stored only in data blocks and not in the action context. Typical actions are small pieces of code designed to establish a desired relation among a few data blocks by examining, and optionally modifying, variables in the data blocks.

Actions can create new data blocks by *instantiation* and new actions by *spawning*. Affinity programs are built by the dynamic creation of a structure of instantiated data blocks and spawned actions with appropriate logical connections. A *root* action, spawned by the kernel, serves as the basis for constructing the rest of the computation.

The code for actions is syntactically similar to that of C++ functions, with a list of formal parameters to which values and references to data blocks are bound when the action is spawned. Unlike functions, new actions may return no value, and are

scheduled for execution independently of the creating action after spawning. After spawning, the only mechanism for communication between actions is modification of shared data blocks.

All actions are peers and have no special relation to actions they spawn. Actions are anonymous; no action identifier is returned to the instantiator, nor used by the kernel. Although an action that instantiates a new data block will initially possess the only reference to it, there is no intrinsic concept of block ownership. If the reference is propagated to other actions, the creator has no special relation to the data block except by a programmer-defined convention.

2.2 Data Blocks

Data blocks are dynamically-created instances of data types, containing virtually all of the state of Affinity computations. Actions have very limited inherent state; data blocks contain the computational state that persists between action activations.

In this implementation, data blocks are typically declared as a C++ `class` (or the "unprotected" variant, `struct`), but any datum explicitly instantiated using the C++ system memory allocator `new`—whether an arithmetic, array or structured type—is a data block. Local data implicitly allocated, i.e., variables of *automatic* storage class, are *not* data blocks.

The Affinity coherence and consistency rules are defined in terms of data blocks rather than the variables they contain. Some concurrent programming models such as PCN [15] and the Dash project's "release consistency" [31] provide two distinct type attributes for variables, one for efficient assignment and a second for synchronization purposes. Affinity's nesting of variables within data blocks can be viewed as a more structured variant of this notion.

Data blocks are the basic entities upon which Affinity is built and operates. The kernel does not require nor use information about the internal structure of the blocks. C++ provides an object model allowing the programmer to structure and manipulate the content of these blocks. The C++ object model is a higher-level abstraction imposed on the data block foundation of the Affinity programming model.

2.3 Triggers and Scheduling

Scheduling of actions for execution is distributed and nondeterministic, although programs may be coded to enforce deterministic effect. Actions may contain special initialization code to be executed one time, and are run at least once.

Actions may set (and clear) *triggers* on individual data blocks. When the state of a data block is changed by an action's successful write, all actions that have set triggers are scheduled for execution. It is guaranteed that an action with a set trigger will execute after a write to the triggering data block, seeing an updated version of the data block contents.

Note that there is not a one-to-one relationship between writes to a data block and action execution. Multiple writes may cause only one subsequent action activation, operating on the "latest" state of that data block, preceding states having been over-

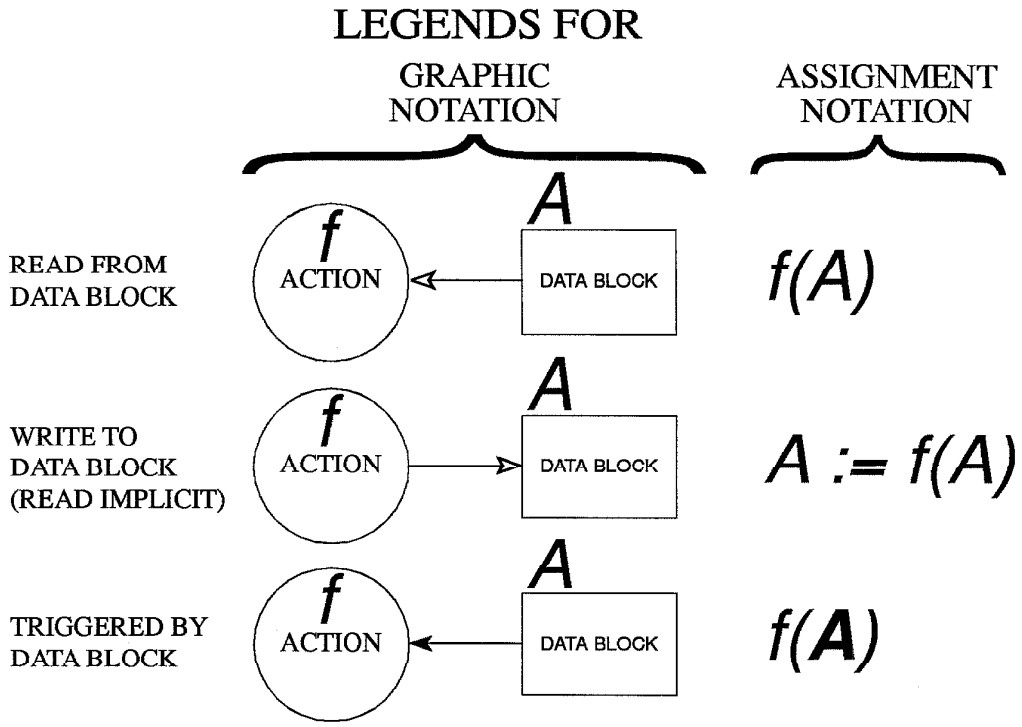

Figure 2.1: Legends for a graphic and a formulaic assignment notation. Each line contains equivalent fragments in both notations. Typically an action also reads from a data block that it writes, but such a read is not specifically represented in the graphic notation. Triggers are indicated by solid arrowheads in the graphic notation, emboldened arguments in the assignment notation.

written. Conversely, it is common to have multiple actions activated after a single data block is written.

The action's context contains a record of data blocks on which triggers are currently set, the *trigger set*. If an action has no triggers set after execution it can be deleted, since there is no requirement for future scheduling.

2.4 Two Schematic Representations

Figure 2.1 introduces two notations for representing the relations between data blocks and actions, to aid in exposition of Affinity program structure. Since this structure is constructed dynamically at runtime, it may require a close reading of the code to identify the input and output blocks of a particular instance of an action. A graphic representation of the same information is more accessible and is also a convenient way to sketch out the problem formulation before writing code. Because actions must communicate via shared data blocks, computations are expressed by specifying the pairwise interactions of actions and blocks. If the number of blocks accessed by an action is limited,

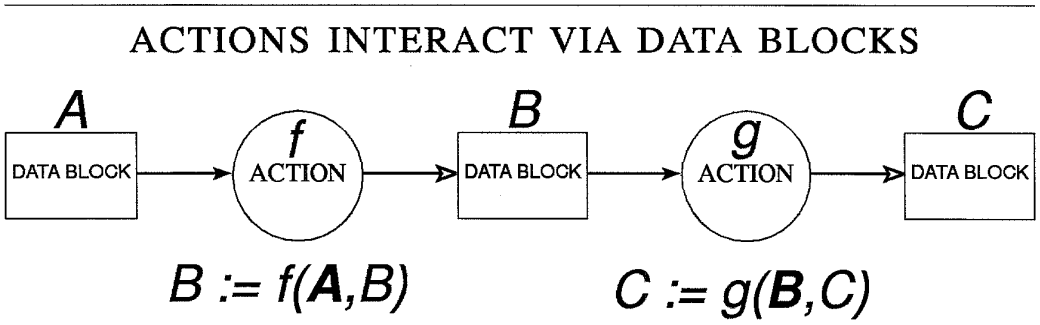

Figure 2.2: Actions interact by modification of shared data blocks. This computational fragment can be represented by either a graphical or formulaic notation.

the computation can be depicted by a set of simple diagrams with easily understood relations. A formulaic assignment notation is a somewhat more compact alternative, better suited to the discussion of Affinity coherence properties.

2.4.1 A Graphic Notation

The left side of Figure 2.1 provides a graphic notation for the most interesting dependencies of Affinity data blocks and actions. The "read from" relation indicates that an action has and uses a reference to a component of a data block, but does not write to the data block. The "write to" relation indicates that the action may modify the block content, often by a read-modify-write access. The "triggered by" relation indicates that a write to the data block will cause the associated action to be scheduled for subsequent execution.

The "read from" arrows show possible information flow. The directed paths and cycles specified by the "write to" and "triggered by" arrows determine the computation's scheduling relations, the reactive analog to traditional control flow structures.

2.4.2 An Atomic Assignment Notation

Affinity actions can be defined in terms of atomic assignment operations, evaluating a function of a set of "input" data blocks and electively making assignments to a set of "output" data blocks. The right side of Figure 2.1 shows the formulaic equivalents of the graphic fragments. An Affinity computation will consist of an aggregation of actions reading from and writing to shared data blocks as shown in Figure 2.2. The properties of the atomic assignment will be defined in Section 2.5.

Neither notation represents the dynamic aspects of Affinity programs, such as data block instantiation, action spawning, or setting triggers. These notations are better suited to describing the "leaf" actions of a computation than the actions higher that build the computational structure. These lower-level structures are typically of greatest detailed interest since they determine the potential concurrency of the program.

2.4.3 Data Block Sharing between Actions

As illustrated in Figure 2.2, Affinity actions interact only indirectly, via data blocks. The sole exception is that actions may instantiate new actions and bind arguments to those instances. The argument lists may contain references to data blocks as well as values; argument list references provide the referential root for sharing data block contents, which may in turn contain more references and values.

Changes to a given data block are automatically propagated throughout the system, but not instantaneously nor indiscriminately. The rules governing the modification and dissemination of data block state (see Sections 2.5 and 2.6) constitute the heart of the Affinity programming model.

2.4.4 Representation of Triggers

Note that both representations of the program fragment in Figure 2.2 show a triggering chain. A write to data block A will cause a subsequent activation of action f. If action f writes to data block B, action g will be scheduled for subsequent execution. A precise definition of triggering is given in Section 2.8.

2.5 Atomicity of Action Execution and Effect

Actions have *atomic execution*: the contents of data blocks visible in an action context are not changed by other actions during its execution. Actions have *atomic effect*: if an action is executed successfully, all its operations have effect. Actions terminate either implicitly by exiting normally or explicitly by *aborting* if no effect is desired. Actions which do not terminate have no effect on computational state. (Nevertheless, a nonterminating action is considered an error, since it may prevent a computation from terminating.)

2.5.1 Action Operational Outcomes

While the only semantically significant outcome of scheduling an action is that it eventually performs an atomic assignment to alter the computational state, it is instructive to touch on the operational aspects of action execution (discussed at length in the next chapter).

Affinity provides no way for an executing action to block or busy-wait for any event. Techniques such as spin-locks and polling are invalid because the data blocks within one action's context are not visibly changed by other actions during its execution. Actions run to completion, either by normal exit or by aborting due to either an explicit request or a system-detected error.

Figure 2.3 shows the three possible outcomes of executing an action. Actions that exit normally may either *succeed* or *fail* in effecting a change in the computational state. Actions exiting normally without performing an assignment succeed, but have no effect except to satisfy a scheduling requirement. Actions which alter the system state by modifying data blocks (explicitly or implicitly) will succeed only if the write-set coherence guarantee (Section 2.6.3) can be satisfied, otherwise they will fail with no effect. This is the "mutex conflict" (mutual exclusion conflict) listed as one cause of

POSSIBLE OPERATIONAL OUTCOMES
OF A ACTION EXECUTION

Success	$B := f(A,B)$	Assignment, establish a condition
	$\phi := f(A,B)$	No assignment, confirm a condition
Failure	$B :\neq f(A,B)$	Abort or mutex conflict or fault

Figure 2.3: Possible operational outcomes of an action execution. A successful action may make assignments to establish some logical condition, or exit without assignment if it confirms that the desired condition already exists. Actions may fail for a variety of reasons, with no effect on the computational state. Successful actions satisfy a triggering event; failed actions will be rescheduled until they succeed.

failure in Figure 2.3. Aborted actions always fail, and have no effect. Some detected faults can be dealt with by forcing an abort.

2.6 Consistency and Concurrency Control

It is important that a final computed result is correct, i.e., consistent with the computational algorithm and the given data. On the other hand, requiring that all intermediate steps be computed with complete knowledge of the whole system state is subversive to the goal of highly concurrent computation. Affinity provides two implicit semantic rules to allow the programmer to express the necessary scopes of computational-state coherence and consistency. One applies to the partitioning of state into data blocks, the other to the actual access made to the data blocks during action execution. It is helpful to augment the assignment notation slightly.

2.6.1 Atomic Assignment

The successful execution of an action f performs an atomic multiple assignment

$$W := f(I)$$

where I is the set of *input* data blocks and W the *write set* for a particular execution of an action. A data block is in I if values in it are accessible to the action by the (transitive) dereferencing of the parameter list. The parameter list is stored in an immutable data block $P \in I$. The action code may transiently modify parameters during execution, but changes will not persist to other activations. When an accessible data block is written by the action f, it is put in the data block write set $W \subseteq I$. It is helpful to define the *read set* as:

$$R \equiv I - W$$

and rewrite the assignment as

$$W := f(W, R)$$

partitioning the input set into disjoint written and unwritten subsets.

It is important to note that the write set is defined by actual, not potential, use. It is defined anew at each distinct execution. The write set is completely defined only at action termination, determined by that particular execution of the action code.

For a particular execution, it is quite possible that $W = \emptyset$, the empty set. The elective choices by the actions of which (if any) of the accessible blocks is written determines which actions will be triggered and consequently scheduled. If all write sets are empty, the computation will become *quiescent*, signifying the termination of at least one phase of the computation.

2.6.2 Data Block Internal Consistency

The atomicity of action execution and effect (Section 2.5) imply that data blocks are modified in discrete steps by (a sequence of) single actions. Each data block will therefore maintain *internal consistency* as the product (after each atomic assignment) of only one action; modified data blocks behave as if they were locked by a successful action during its execution.

2.6.3 Data Block Write-Set Coherence

Affinity guarantees that an action that writes to a data block will modify the current version of the data. The programmer can therefore induce coherence among a set of data blocks by writing to all members of the set. This mechanism of *write-set coherence*, in combination with the *internal consistency* of data within a single data block, provides an adequate basis for useful computation.

Merely reading a data block does not guarantee that the current version of the data is the version visible to an action: a data block may be *stale*. If currency is required, the action may add data blocks to the write set by *touching* them, i.e., by performing a nondestructive write. This technique tends to reduce concurrency, and should be employed sparingly; its use may indicate poor program design.

2.6.4 Data Block Versions

An action assignment operation can be viewed as performing a functional evaluation of one version of the values of its input data blocks and creating a new version of the data blocks that it writes. We can number the sequence of states of any given data block as it is written by successful actions. Data blocks that are read but not written do not have new versions created.

Define the version number of a data block $w \in W$ as the number of times it has appeared in the output set of a successful action after its instantiation, and define a corresponding version function $V(w) \in Z$ which returns the version number. We add a version marking to the atomic assignment statement

$$W' := f(W, R)$$

where each write-set data block $w' \in W'$ is the next version of $w \in W$, i.e.,

$$V(w') = V(w) + 1.$$

The *current* version of a data block is the maximal version extant in the system.

We can restate the axioms of Sections 2.6.2 and 2.6.3 as

Internal Consistency: the contents of data blocks are observable only for discrete versions with integral version numbers.

Write-set Coherence: $\forall\ w \in W$ the version number $V(w)$ is incremented atomically upon the success of an action's multiple assignment.

Since only one action may write to a particular version and a successful assignment increments the version by exactly one, any $w' \in W'$ will be the current version at the instant of the atomic multiple assignment. Figure 2.4 graphically illustrates the consistency and coherence rules introduced in Sections 2.6.2 and 2.6.3 and restated here.

2.7 Coherence and Consistency

The terms *coherence* and *consistency* are sometimes used interchangably. In the Affinity computational model, *coherence* is a statement that the versions of data blocks used by an action are the current versions. Coherence derives from the atomicity of action multiple assignment and the write-set coherence rule.

Consistency is a more subtle thing, the satisfaction of some required condition as defined by the program's logic. Data blocks are internally consistent because each version results from a single action's assignment. A read set is not guaranteed to be coherent; some data blocks visible to an action may be stale (not the current version). Incoherence, a "physical" property natural to multicomputers, need not produce inconsistency, a logical property. A properly coded program will be able to tolerate read-set incoherence without effecting an inconsistent result, a logical error.

Read-set incoherence is allowed in the computational model to improve concurrency in physically large computers by relaxing synchronization constraints. Write-set incoherence, which would greatly complicate the difficulty of computing a meaningful result, is disallowed.

2.8 Triggering Defined

The goal of the triggering mechanism for action scheduling is to allow the programmer to ignore some of the problems stemming from read-set incoherence without compromising the quality of the final computational result. When an action is triggered by a write to a data block, it is guaranteed that the action will eventually execute seeing an equally- or more-recent version of that data block in its input set. Formally, an assignment

$$W' := f(I)$$

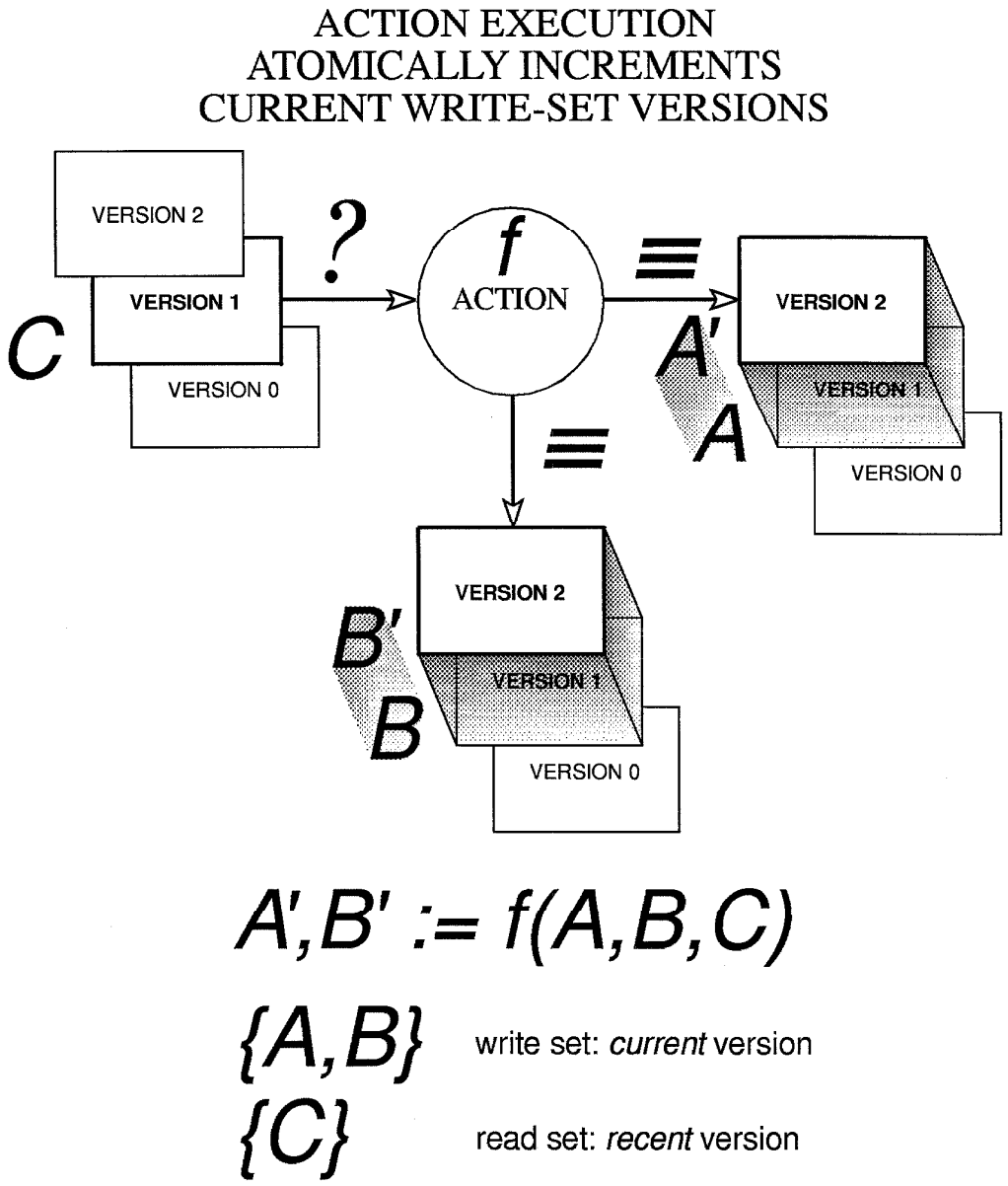

Figure 2.4: A successful action will always modify the current version of data blocks in its write set; no such guarantee applies to data blocks which are in the read set only. A new version is produced by each successful atomic assignment to a write-set data block. The write-set blocks $\{A, B\}$ are the current versions when the function is evaluated. After the assignment succeeds, the write-set versions are incremented and the new versions $\{A', B'\}$ become current.

with data block $w' \in W'$ implies that eventually every action with a trigger set on data block w will succeed with w'' in its input set and

$$V(w'') \geq V(w')$$

that is,

$$W' := f(I) \quad \longrightarrow \quad W''' := g(I'') \text{ with } w' \in W', \; w'' \in I''.$$

Observation of system quiescence implies that all actions have evaluated the current version of all data blocks in their trigger set.

2.9 Quiescence and Validity of Results

A computation may define termination by some logical test of the state of its own data objects, e.g., seeing that some determinate number of tasks have been performed, or by using system-provided information to observe that all triggered actions have executed without triggering further actions, i.e, that the computation is *quiescent*. The observation of system quiescence provides a global "barrier synchronization" that the program logic can interpret as termination of a part or the whole of a computation.

If the actions have been coded to establish particular relations between their triggering input data blocks and their output blocks, those relations are known to hold at quiescence, since all the actions will have evaluated the most recent version of the input data and confirmed the relation without assignment. (To be precise, an assignment to a data block with set triggers.) The triggering mechanism in this way guarantees that significant transient inconsistencies in the computational state (if any) have been eliminated. Results from a nonterminating computation are accessible provided that the component actions terminate, but establishing their correctness requires more detailed reasoning.

2.9.1 A Semi-Coherent Quiz

Figure 2.5 specifies a computational fragment complex enough to illustrate the basic Affinity rules. The figure shows both a graphic representation of the structural composition of three actions and some pseudo-code specifying their behavior. It is understood that write-set data block versions are incremented, so the pseudo-code omits the "prime" version markings of the previous section.

Initially data blocks A and C are zero and data block B contains a 1. The 1 represents a token which the two actions f and g will compete to acquire. There is only one token, which shouldn't be duplicated. The actions must therefore zero B if they copy the 1 to their private variables A or C, respectively. Given the initialization, the action code can simply exchange values. The multiple assignment statements do this by first evaluating the expressions on the right-hand side of the assignment operator and then assigning new values to the left-hand side. The caption poses three questions which we will consider sequentially.

For the first question, "how many tokens are visible to each action?" there are several correct answers, and, equally important, some incorrect ones. Let us consider the perspective of actions f and g. While $B = 1$ both actions will attempt to write to it

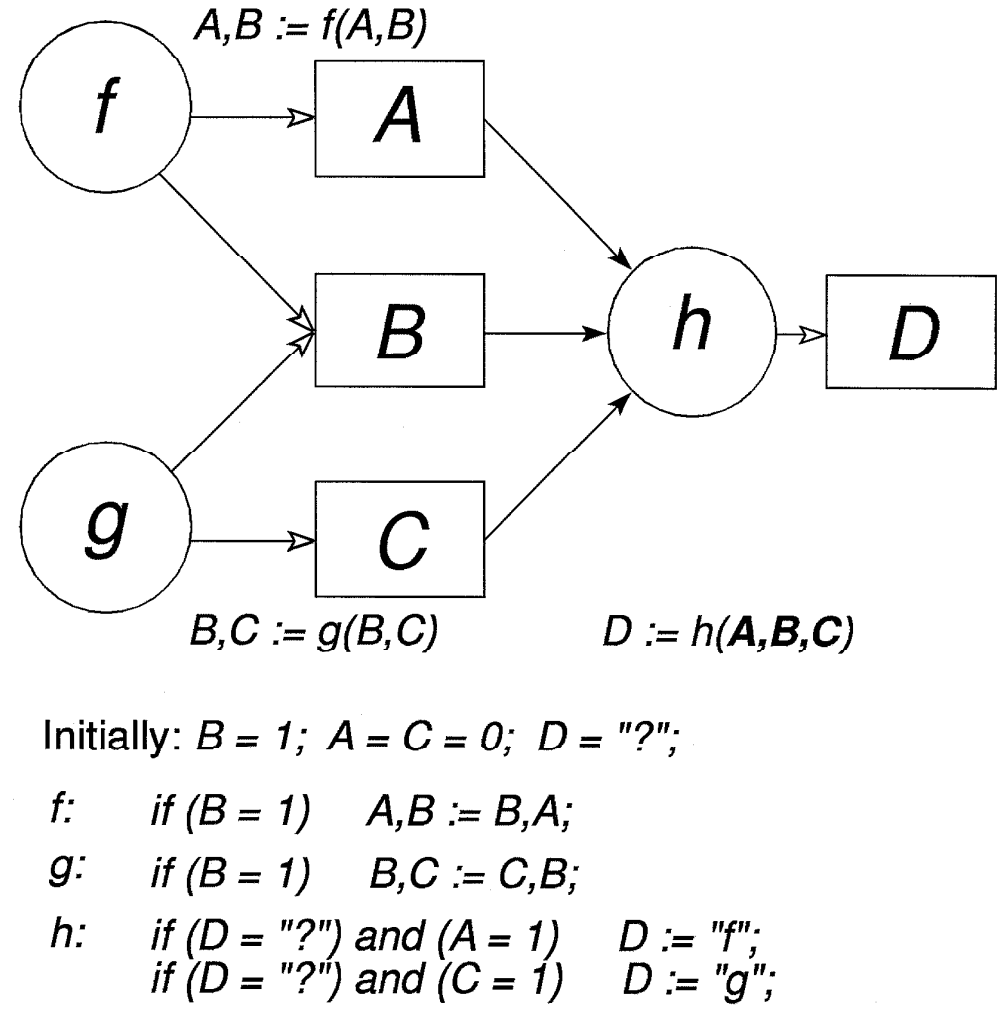

Figure 2.5: Write-set coherence and read-set incoherence. Two actions compete for a single token; a third observes passively. How many tokens are visible to each action? How many times does action h execute? How many times does action h make an assignment to D?

as part of a swap. Write-set coherence enforces mutual exclusion. Only one of the two actions will succeed in acquiring the token; the other will subsequently succeed without making an assignment when it views B as zero. The winner sees one 1, which it will swap. The loser sees no 1's. (Operationally, the losing action might have seen $B = 1$, but the attempted exchange assignment would subsequently have failed. Semantically, it makes no difference whether it executed and failed or ran only after the other action had acquired the token.)

The possibilities for h are more numerous. Since h is triggered by all of the blocks, we know that it will coherently reflect their current state at quiescence: h will see one 1 at its final activation. What about transient incoherence? Action h may run before the others, seeing $B = 1$. It may see the change in one of the two blocks swapped by the winner before it sees the other, possibly seeing the token in either two blocks or none before the incoherence is removed. So the count of tokens seen by h will be some sequence of the form $(0|1|2)^*$ 1, where the $*$ means zero or more such digits and | is an alternative. Three is a wrong answer though. The incoherence of the read set may cause h to jumble various views of actual events, but only one competitor can acquire the token. A count of three would require that the token appear in both A and C, which is impossible.

How many times does action h execute? At least once subsequent to the success of one of the actions competing for the token. The Affinity model doesn't set an upper limit.

The reader may be pleasantly surprised by the strength of the answers given for actions f and g and disquieted by the apparent weakness of the answers given for h. The distinction between incoherence and inconsistency can be made in this example. Action h's view of the contents of A, B, C may be incoherent. Is it inconsistent? Not with proper interpretation. If we ask "which action got the token?" h can always give a correct answer ("f," "g," or "?"(don't know yet)) given knowledge of the logic of the other actions. Only incorrect coding of h will translate read-set incoherence into an erroneous result. To answer the third question, the test of output block D ensures that the program for h writes the result only once.

2.9.2 Initialization and Races

In the example above, we avoided initialization-race concerns by specifying that all the actions of Figure 2.5 were spawned by a common parent action. Atomicity of effect applies to data-block instantiation, action spawning, and trigger setting, as well as to explicit assignment. For the example we required that the initial values be set in the parent action, guaranteeing that the initial state will hold before any spawned child actions execute.

2.10 Action Address Spaces and Contexts

Actions may have distinct virtual address spaces. Affinity provides a special mechanism for declaring *portable pointer* types to data blocks and components of data blocks (Section 3.3.1) that allows references to be shared between different action contexts. Standard pointer types, e.g., `void *`, are not generally meaningful outside of one action

context; any use of them in a data object or in a formal parameter list declaration is an error. Portable pointers are required in their place.

Action code may access any data block to which it has a reference. References (in the form of portable pointers) may be passed in argument lists or via shared data blocks. Use of a reference (*dereferencing*) causes the named data block to be mapped into the virtual address space of the action so that it may be accessed by the usual pointer idioms. Data blocks mapped in the address space of more than one action are logically shared. The scope and degree of overlap of action address spaces may be tailored to the particular computation.

2.10.1 Action Essential Context

Affinity actions have very limited *essential* (semantically significant) context. Essential action context comprises an argument list (containing the values and references bound at action instantiation), one bit of state indicating whether the action has run at least once (to support initialization code), and references to the code and (optional) trigger-set data blocks. This essential state is stored in data blocks that the programmer need not explicitly consider. Since this state is in data blocks, it does not complicate the programming model. The relative obscurity of these blocks is simply a convenience for the programmer.

During action execution, additional *transient* context is required. Transient context may include processor state, stack variables and mappings of dereferenced data blocks. Neither processor nor stack state is preserved between action executions. The degree to which block mappings are retained between activations is an implementation-efficiency issue.

2.11 Logical and Effective Concurrencies

The effective concurrency of an Affinity program will be determined by the physical concurrency of the hardware (i.e., the number of processing nodes) and the program design, in particular the data structures employed. The implicit write-set consistency rule induces a form of mutual exclusion on actions which can reduce the effective concurrency of a computation. What may be surprising is that the same mechanism can be used to increase concurrency.

2.11.1 Reduced Concurrency by Action Failure

Programs in which multiple actions frequently write to a shared data block (the *multiple-writers* case) may exhibit limited effective concurrency and high action failure rates. Designing a computation's shared data structures to avoid excessive coherence and consequent serialization is the Affinity programmer's goal.

2.11.2 Increased Concurrency by Action Cloning

One might assume that the number of actions spawned by the program imposes an upper limit on the potential concurrency of a computation, but this is not necessarily so. While the specific formulation of the code in terms of actions will substantially affect

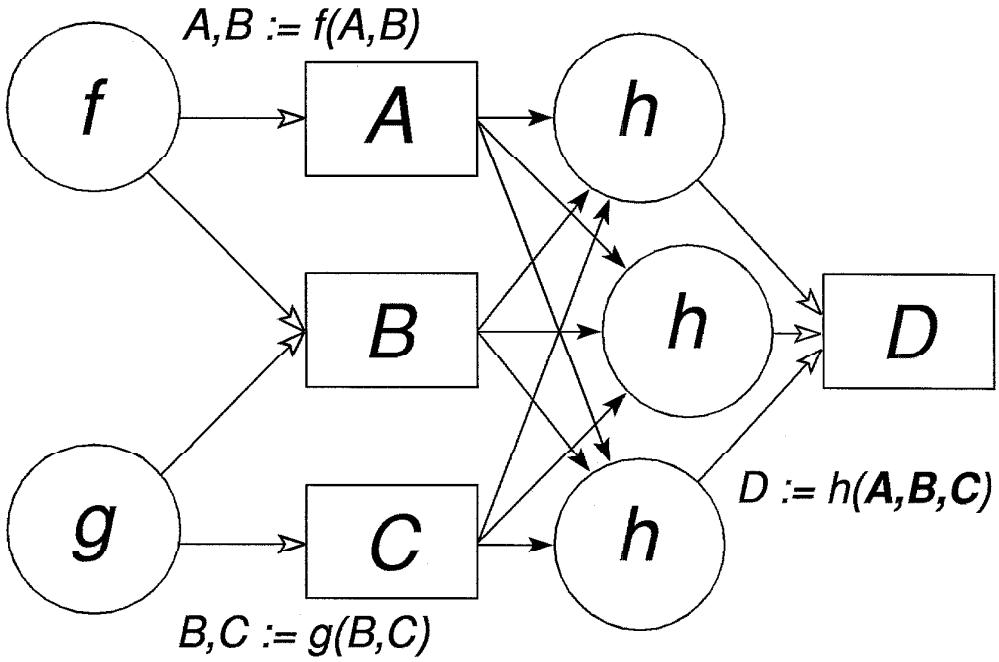

Figure 2.6: This program is the same as that of Figure 2.5. The cloning of action h has no semantically significant effect, and requires no alteration of the code. The other actions could likewise be replicated without semantic effect. Cloning can be transparent to the programmer. An implementation of the programming model can alter performance and reliability characteristics by its action replication policies.

the concurrency achievable on a given machine, Affinity semantics allow the kernel to replicate action instances without altering the program logic, so the number of actions may exceed the number explicitly spawned.

If we reconsider the example of Figure 2.5, we recall that we could not precisely specify the number of times action h would execute, nor exactly what its environment, the read set, would look like. The weakness of the specification allows us the freedom to introduce multiple identical copies, called *clones*, without altering the program semantics, as shown in Figure 2.6. The write-set coherence rule means that no matter how many clones are extant, the answer will be written to D only once. Because the clonal actions do not contain any state hidden within, they have no private history. There is no semantic distinction between a sequence of executions by a single action and concurrent execution by multiple clones of that action. In general, actions have no singular identity.

2.11.3 Action Cloning and Fault Tolerance

Action failure is semantically correct, whatever its cause. A failure due to detection of a transient hardware error (e.g., a parity error) need not be treated any differently than one due to a mutual-exclusion data-block-access conflict, given that the action can be rescheduled. Even a severe hardware fault that causes a node to fail and stop may not be significant provided that an action has clones on other nodes. Translating this interesting fault-tolerant aspect of the model into a genuinely fault-tolerant implementation would require comprehensive care; the subject is discussed further in Section 5.7.

Chapter 3

Introductory Examples

The previous chapter introduced the Affinity computational model in a relatively abstract manner. This model differs considerably from more common programming styles based on processes with inherent state. The purpose of this chapter is to demonstrate that the combination of properties and features peculiar to Affinity yields an expressive notation for concurrent programming.

Several programs are presented in this chapter to provide operational examples of Affinity programming. The initial examples are quite short to allow detailed explanation; larger programs are presented later in a more summary fashion.

The first example programs are tutorial in some detail.

hello1 (Section 3.2) shows how actions are defined and spawned.

hello2 (Section 3.3) shows how private data blocks are created and used.

hello3 (Section 3.4) shows the creation and use of a global data block, with the program logic enforcing sequential writing to a shared variable.

chop (Section 3.5) demonstrates Affinity's implicit mutual exclusion of simultaneous multiple writers.

Next a standard concurrent communications problem is examined in some detail.

prodcon1 (Section 3.6) demonstrates use of a bounded buffer class to solve a single-producer/single-consumer problem.

prodcon2 (Section 3.6.5) shows how Affinity's implicit mutual exclusion allows multiple competing consumers to use the same buffer object without requiring any change in the code.

symmbuffer (Section 3.7) is a symmetric generalization of the buffer class usable as general-purpose program building block for bidirectionally-communicating actions.

The impatient reader may wish to skip toward the end to glance at some more substantial programs that produce a computational result.

Matrix multiply (Section 3.8) allocates and initializes a square dense matrix constructed as a vector of vectors and produces a product matrix. The same code can be compiled to give either a concurrent or a sequential program.

APSP (Section 3.9) is a graph shortest-path algorithm that shows how an Affinity concurrent program can be expressed with a natural simplicity that compares favorably with sequential algorithms. The matrix and vector types of the previous example are reused and extended.

Finer-Grain APSP (Section 3.9.2) is a variant that shows that a modular substitution of core code with a quite different implementation requires minimal or no change to the remainder of the program code.

These examples have been compiled and run on the S/2010 implementation of Affinity.

3.1 Language Restrictions

The language of the examples is a restricted subset of C++ [52, 18] with a few predeclared classes, macros, and system-interface functions, which will be introduced as encountered. The restrictions include: no global variables, no `static` local variables, and no standard pointers in shared data structures or action parameter lists. A predefined macro supports declaration of typed *portable pointers* that may be used in any context.

To minimize inessential detail and to reduce the cognitive distraction for readers unfamiliar with C++, the example code is somewhat simplified. Specifically, it neglects C++ information-hiding principles by declaring objects using the `struct` rather than the more common `class` keyword, making all data members freely accessible. Likewise, the `const` keyword should properly be widely used, but is not.

The distinction between Affinity data blocks and C++ objects drawn below (Section 3.6.1) is unimportant for the initial examples. In the following text the term *object* is generally used as a preferred synonym, except when the detailed data structure is the specific interest. C++ *member functions* are called *methods*.

3.2 A Multiple Action Example

A minimal example (**hello1**) with multiple concurrent actions is listed in Figure 3.1. The program creates several new actions, each of which prints one message to the console.

The included file "affinity.h" contains declarations of system data types, macros, and kernel-interface functions. The only action declared in this program is called "printer," which has a formal parameter list consisting of a single integer, that is passed by value in C++ (as in C) and requires no special treatment. The `declare_action` macro accepts two arguments: the action's name and a formal argument list, which is syntactically identical to a C++ function argument list. The body of each action is syntactically that of a C++ function.

The `root()` action corresponds to the `main()` function of most C++ or C programs. When an Affinity program is run, the `root()` code is loaded and executed on some multicomputer node. It is an Affinity action that executes only once, creating other

```
//  Hello example

#define NPRINTERS       4

#include <affinity.h>

declare_action(printer,(int my_id))
{
  printf("Hello world from printer %d\n",my_id);
}

root()
{
  int   i;
  for (i = 0; i < NPRINTERS; i++) {
    spawn(printer(i));
  }
}
```

Figure 3.1: Multiple-action **hello1** program

actions and data objects and providing the logical binding between them. The current implementation does not provide any arguments to `root()`.

3.2.1 Spawning New Actions

In this trivial example, the program creates only actions, not data objects. The root action contains a loop that instantiates several copies of the "printer" action. The body of the loop is a predefined macro, **spawn**, which takes one argument: an action name with a list of arguments to be bound to the action's formal parameter list. The parameter values are copied by the kernel, and are fixed for the life of the action instance. At each activation of the action the values of the parameters are those bound when the **spawn** was executed and do not reflect any subsequent changes of state. An action may modify a parameter-list variable during execution, but changes will not persist to a subsequent activation: the originally bound value will reappear.

No value is returned by the **spawn** expression. There is no special relation between the parent (spawning) action and child (spawned) action after spawning; the parent does not receive any reference to the child. After spawning all actions are peers.

3.2.2 Operation of the `hello1` Program

The operation of the program is straightforward. The root action instantiates four "printer" actions and exits. Each "printer" action executes one time on some multicomputer node, sending one line of text to the console, as seen in Figure 3.2.

```
0,711: Hello world from printer 0
1,711: Hello world from printer 1
2,711: Hello world from printer 2
3,711: Hello world from printer 3
```

Figure 3.2: Console output of **hello1**

The S/2010 Affinity implementation uses a console emulation service provided by the Cosmic Environment (CE) [53]. CE is a message-passing programming environment developed at Caltech, which runs on a number of multicomputer systems and networked workstations. The initial "0,711: " prefixed to the first output line is added by the CE console server process, indicating that the output is from node 0 and process ID 711, which is the process ID used by the Affinity kernel in its communications with the CE. (In this chapter's examples, the mapping of actions to multicomputer nodes is sequential, with one action per node, and actions are not relocated after initial placement.) In this particular example the concurrent actions happen to produce an ordered output sequence; that is, the line from "printer 0" appeared on the console before that from "printer 1." This is not generally the case; the order of action scheduling is nondeterminate unless the programmer explicitly implements some scheme to induce action sequencing.

3.2.3 Printing to the Console

To maintain Affinity action atomicity, the Affinity `printf` function ensures that console output from a failing action is not printed. A kernel-spawned printer action is triggered by `write` operations to a buffer similar to that of Section 3.6. It simply reads from the buffer, invoking a direct message-sending system service (`dprintf()`) for each line (see Section 4.13.1).

3.3 An Example with Data Objects

The example program **hello2** listed in Figure 3.3 is an elaboration of the previous example. This program creates several new actions, each of which prints multiple messages to the console. If the sole purpose were to print multiple output lines, we would simply use a `for` loop, but we want to illustrate how an action can be activated in response to (*triggered by*) changes in data objects. For this reason we code the action to print only one output line per activation and cause each action to be executed multiple times.

Since the "printer" actions contain no persistent state, we need to instantiate a data object to record the number of lines remaining to be printed. This is done by the root action in the line

```
*(lines_to_print = new int) = NLINES;
```

which instantiates a new data object, an integer, in standard C++ fashion. Typical data objects are more complex, defined as a `struct` or `class`, but fundamental types (and

```
// Retriggering hello example

#define NPRINTERS       4
#define NLINES          3

#include <affinity.h>

declare_pointer(int,int_p);

declare_action(printer,(int_p lines_left_p, int my_id))
{
  int   &lines_left = *lines_left_p;
  lines_left_p.set_trigger();
  if (lines_left)
    printf("Hello world from printer %d, %d lines left\n",
           my_id,--lines_left);
}

root()
{
  int   i, *lines_to_print;
  for (i = 0; i < NPRINTERS; i++) {
    *(lines_to_print = new int) = NLINES;
    spawn(printer(lines_to_print,i));
  }
}
```

Figure 3.3: **hello2**, showing data instantiation and triggers

arrays of fundamental types) are also useful. The standard C++ operator `new`, which dynamically allocates storage (here, for an integer), works as expected under Affinity. In addition, Affinity guarantees that the returned storage is initially zeroed. (So `new int` is similar to the C expression `(int *) calloc(sizeof(int))`.)

In this example a distinct data object (an `int`) is instantiated on each pass through the `for` loop and assigned an initial value of `NLINES`, here 4. Note that the "printer" action parameter list now contains an `int *`, `lines_to_print`, in addition to the `int i` of the previous example. In Affinity each explicit dynamic memory request (`new`) to the default system allocator will instantiate a data object that may be shared with other actions by providing a reference to it.

3.3.1 Portable Pointer Declaration

Near the top of the code in Figure 3.3, the `declare_pointer` macro is used to declare the type `int_p` as a portable pointer to `int`s, similar to `int *`. This is a C++ class

declaration (similar to a C *typedef*) and not an instance of the class; `declare_pointer` specifies a type, not a variable. The formal argument list of the action "printer" now contains an `int_p` declaration, illustrating the Affinity rule that pointer types must be declared with the `declare_pointer` macro before use as formal arguments to actions (or use in persistent (non-stack) data objects). C++ forces automatic coercion of pointer types to match the formal arguments, providing a mechanism to support references (*portable pointers*) usable in all action contexts throughout the multicomputer.

The first line in the body of action "printer"

```
int &lines_left = *lines_left_p;
```

may seem exotic to C programmers, but is simply a bit of C++ "boilerplate" to allow `lines_left` to be established as a more readable alias for `*lines_left_p`.

Another novelty is the presence of a call to the `set_trigger()` method defined for portable pointers by the `declare_pointer` macro. `set_trigger()` causes the action to be scheduled for execution whenever the data object referred to is written by any action. Repeated "setting" of a trigger on an object is redundant but harmless if somewhat inefficient; in this example, the `set_trigger()` method is called each time the action executes.

Actions will always be activated (run) at least once; but will be deleted after that initial activation unless at least one trigger is set on some data object. Actions with a trigger set are guaranteed to be activated subsequent to a modification of a "triggering" data object.

The action decrements the `lines_left` counter each time it is activated, stopping when it reaches zero. When the `lines_left` data object is modified by the action, it triggers another subsequent activation of the same action. This "externalized" loop counter causes a strict sequentiality of the output from a single action, but the other actions, each with a distinct counter object, are completely independent.

The output log of this example (Figure 3.4) shows a particular interleaving of four distinct print sequences. Unlike the previous example, there is no global order: the line from "printer 2" repeatedly arrives at the console before that of "printer 1." The repetition of this pattern is a chance artifact of the sequentializing effect of printing to the single, global, console.

When the action writes to its (in this case, sole) triggering object it causes all actions with triggers set (in this case, one, itself) to be scheduled for activation. Since it stops modifying the counter object `lines_left` when it reaches zero, the triggering stops and the action is no longer scheduled for activation. This particular program does not detect termination of the computation as a whole, but all the actions do stop printing to the console after the assigned number of lines.

As written, the computation becomes *quiescent*, that is, demonstrates no activity, but the "printer" actions and data objects are still extant in the system. The programmer may explicitly force the destruction of actions and data objects when they are no longer useful. There is a `clear_trigger()` portable-pointer method that negates the effect of `set_trigger()`. If an action has no triggers set after execution it will be deleted, since there is no requirement for future scheduling. (The `root()` action of the example is, therefore, activated only once.) If we wanted the action to be deleted after it was done printing, we could add code to clear the sole trigger.

```
0,711: Hello world from printer 0, 2 lines left
2,711: Hello world from printer 2, 2 lines left
1,711: Hello world from printer 1, 2 lines left
3,711: Hello world from printer 3, 2 lines left
0,711: Hello world from printer 0, 1 lines left
2,711: Hello world from printer 2, 1 lines left
1,711: Hello world from printer 1, 1 lines left
3,711: Hello world from printer 3, 1 lines left
0,711: Hello world from printer 0, 0 lines left
2,711: Hello world from printer 2, 0 lines left
1,711: Hello world from printer 1, 0 lines left
3,711: Hello world from printer 3, 0 lines left
```

Figure 3.4: Console output of **hello2** showing one particular interleaving of four ordered printing sequences. The printing order within each "printer" output sequence is guaranteed; the relationship between "printer" sequences is chance.

```
if (!lines_left)
  lines_left_p.clear_trigger();
```

As one might expect, the standard C++ `delete` operator can be used to force the destruction of objects allocated by **new**.

The essential computational structure of the **hello2** program listed in Figure 3.3 is sketched graphically in Figure 3.5. This diagram omits the transient root action that instantiates the rest of the computation. Each "printer" action has access to only one of the data blocks containing the `lines_left` variables. The cycle of "write to" and "triggered by" arrows between each "printer" action and its associated data block indicates a close coupling due to the somewhat contrived use of an "externalized" loop counter. This is the simplest case of a scheduling cycle. The complete independence of the four actions (after spawning) is visually evident from the lack of any relations between the action/block pairs.

3.4 A Shared Data Object Example

The example program (**hello3**) diagrammed in Figure 3.6 and listed in Figure 3.7 is an elaboration of the previous example. This program introduces a shared data object (`current printer`, an `int`) to impose a total ordering on the sequence of printed output lines. Each action checks the value of the `current_printer` variable to see if it is its turn to print. If so, it prints a line and decrements its `lines_left` counter, triggering itself for the requested number of printing operations. When the `lines_left` counter is zero, the shared variable `current_printer` is decremented. This causes all extant `printer` actions to be activated to reevaluate whether their turn has come. The lines

```
lines_left_p.clear_trigger();
current_printer_p.clear_trigger();
```

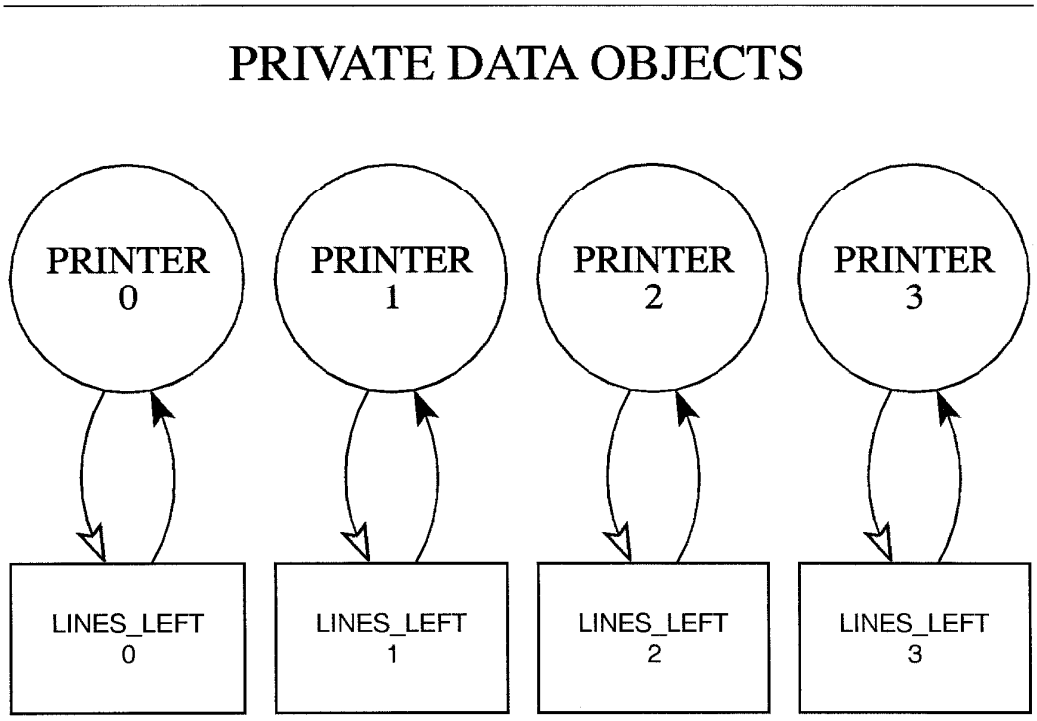

Figure 3.5: **hello2** program

cause the action to be deleted after it is finished printing.

3.4.1 Private and Shared Objects

A data object can be accessed by any action that has a reference to it. The distinction between private and shared, or, for that matter, global, objects is created only by program convention and usage. (The use of "private" here is distinct from C++ use of the term as a class-access-control keyword.) The **hello3** program contains one global object (`current_printer`) and one object that is in effect private to each "printer" action because the other actions have no reference to it.

3.5 Two Dining Philosopher Programs

Dijkstra's dining philosopher problem is a classic example of the difficulties resulting from concurrent access to shared resources [17]. Our Chinese variant of the problem [16] uses chopsticks instead of forks.

`N` philosophers are seated at a circular table on which are `N` chopsticks, one between each pair of philosophers, as sketched in Figure 3.9. Each philosopher wishes to eat `NBITES` from a bowl of rice before her, one bite at a time, returning the chopsticks to the table between bites in order to more perfectly contemplate the virtues of a good meal. At any instant a given chopstick can be used by only one philosopher.

SHARED AND PRIVATE DATA OBJECTS

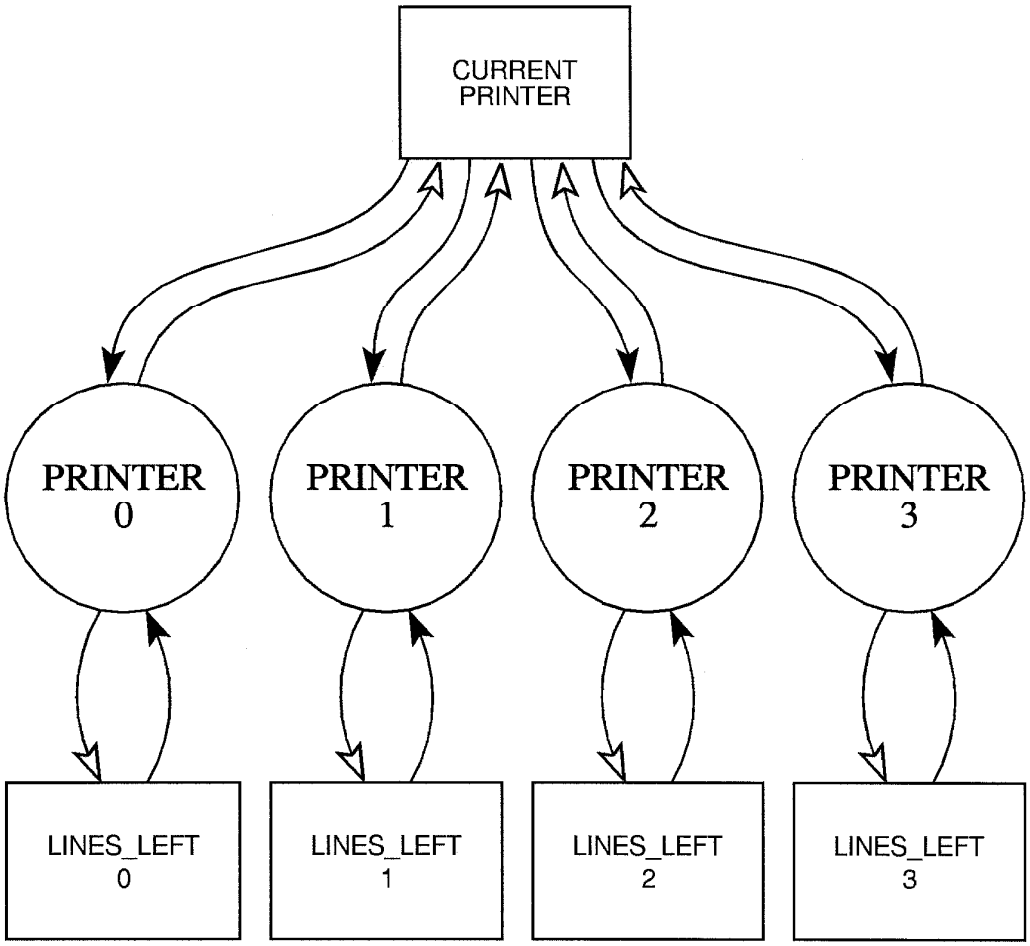

Figure 3.6: **hello3** program

The points of interest in this example under Affinity are somewhat different than the usual textbook discussion. The program is useful to illustrate atomicity of effect and mutual exclusion by and between action activations. Traditionally, this problem is used to demonstrate two problems that may arise in other environments. If each philosopher were to simultaneously pick up the chopstick to her left, then attempt to pick up the chopstick to her right, deadlock would result since all the chopsticks would be in use after the first operation. In a conventional sequential language, executed on a shared-memory multiprocessor, the problem of deadlock arises from the sequential and nonatomic acquisition of resources (e.g., chopsticks). The Affinity programming model

```
// Sequentialized hello example
#define NPRINTERS       4
#define NLINES          3
#include <affinity.h>
declare_pointer(int,int_p);

declare_action(
printer,(int_p lines_left_p, int_p current_printer_p, int my_id))
{
  int   &lines_left = *lines_left_p;
  int   &current_printer = *current_printer_p;
  lines_left_p.set_trigger();
  current_printer_p.set_trigger();
  if (current_printer == my_id)
    if (lines_left)
      printf("Hello world from printer %d, %d lines left\n",
             my_id,--lines_left);
    else {
      current_printer++;
      lines_left_p.clear_trigger();
      current_printer_p.clear_trigger();
    }
}

root()
{
  int   i, *lines_to_print;
  int   *current_printer = new int;     // zero by default
  for (i = 0; i < NPRINTERS; i++) {
    *(lines_to_print = new int) = NLINES;
    spawn(printer(lines_to_print,current_printer,i));
  }
}
```

Figure 3.7: **hello3**, showing a shared data object

guarantees that operations performed during a single action activation are atomic in effect, so several statements in the action code may modify distinct data objects (in effect asserting temporary ownership) in any logically correct manner without concern for deadlock arising from incremental resource allocation. Logical deadlock is still possible if an incremental resource allocation is performed by a sequence of action activations, but this is not a natural programming result. In Affinity the characteristic programming issues are consistency and concurrency rather than avoidance of deadlock. The second problem is starvation: some philosopher could be denied access to two chopsticks for

```
0,711: Hello world from printer 0, 2 lines left
0,711: Hello world from printer 0, 1 lines left
0,711: Hello world from printer 0, 0 lines left
1,711: Hello world from printer 1, 2 lines left
1,711: Hello world from printer 1, 1 lines left
1,711: Hello world from printer 1, 0 lines left
2,711: Hello world from printer 2, 2 lines left
2,711: Hello world from printer 2, 1 lines left
2,711: Hello world from printer 2, 0 lines left
3,711: Hello world from printer 3, 2 lines left
3,711: Hello world from printer 3, 1 lines left
3,711: Hello world from printer 3, 0 lines left
```

Figure 3.8: Console output of **hello3** showing total ordering by use of a globally-shared data object

an indefinite time. Since we have posed this particular problem in a finite form, the issue of fairness is of reduced interest here, but in general, since Affinity makes no effort to ensure fairness of action scheduling, the programmer may need to write code to ensure fairness if it is required. Practically, fairness is more of a problem when mutually exclusive modifications of shared data objects cause frequent actual write conflicts (as in this example).

In a distributed memory multicomputer it is unlikely that all philosopher actions can simultaneously observe the same state of chopsticks. Let us say that we indicate ownership of a chopstick by placing an identifying value ("chop") in an "owner" field in a data structure. To ensure a consistent state of the chopstick ownership we must ensure mutually exclusive access by multiple (two) writers. Expressing a solution to this problem in Affinity is quite straightforward, although necessarily longer than the previous examples.

Figure 3.10 contains the declaration of the `chopstick` data type, which defines a variable (`owner`) to record which philosopher action is its current owner and four methods to sense, acquire and release ownership. The line

```
chopstick()     { release(); }
```

is a simple `constructor` function which initializes the `owner` field to an unused ID when it is instantiated. `SHOW` can be defined to be `printf` or `bit_bucket`, a predefined function which does nothing, depending on whether a printed trace of the execution is wanted.

Figure 3.12 defines the "philosopher" action; Figure 3.11 shows how these actions are instantiated and logically connected. In Figure 3.11 the line

```
chopstick_p *chop_dict = new chopstick_p[NPHILOSOPHERS];
```

instantiates a data object which is an array of references. This is a "dictionary" of the portable pointers to each chopstick. The line

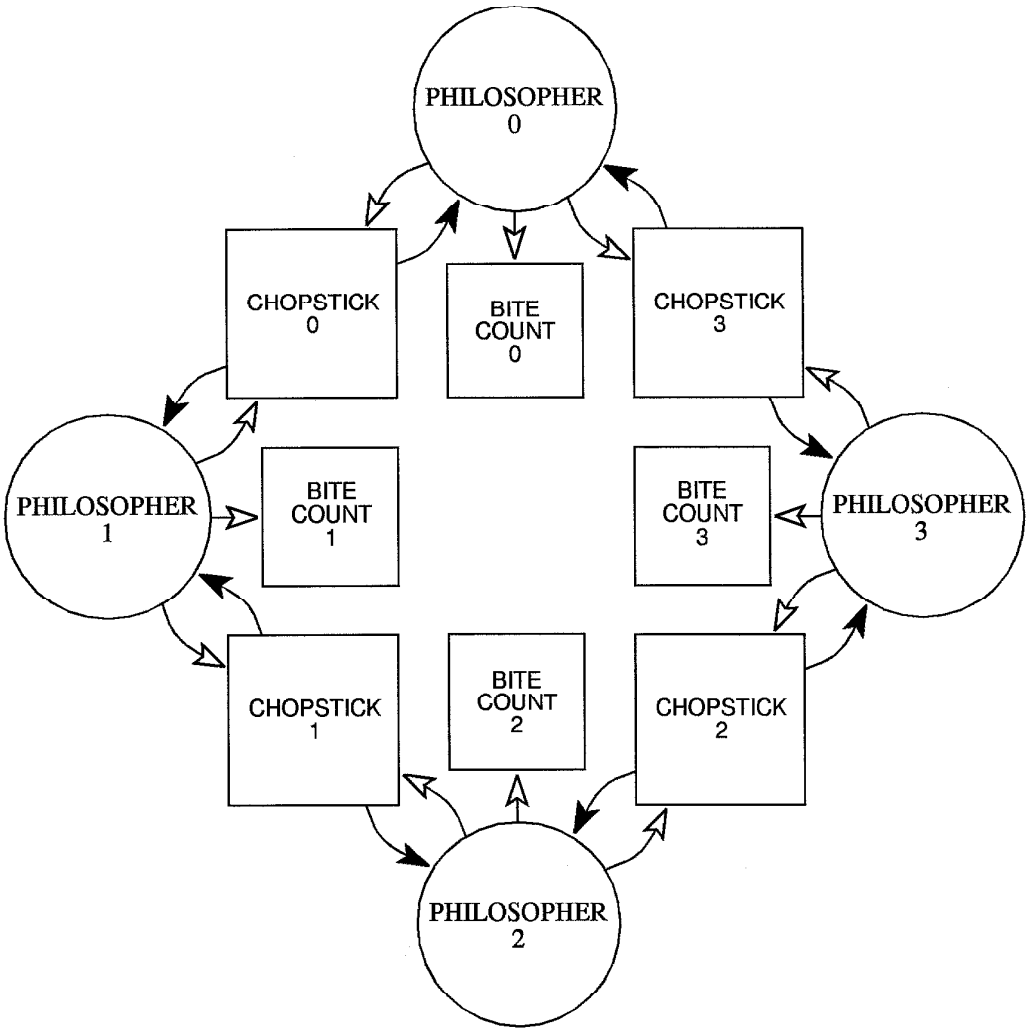

Figure 3.9: Dining Philosophers program

```
// Dining philosophers example

#define N               8
#define NBITES          4
#define SHOW            bit_bucket

#include <affinity.h>

struct  chopstick       {
  int   owner;                      // ID of current owner
        chopstick()     { release(); }
  int   unused()        { return (owner == -1); }
  int   mine(int id)    { return (owner == id); }
  void  get(int id)     { owner = id; }
  void  release()       { owner = -1; }
};
declare_pointer(chopstick,chopstick_p);
declare_pointer(int,int_p);
```

Figure 3.10: Dining Philosophers solution **chop** — data structure

```
chop_dict[i] = new chopstick;
```

instantiates new chopsticks and records the reference for each in the dictionary array. Two chopstick portable pointers (of type `chopstick_p`) are passed to each action upon action instantiation (by **spawn**), corresponding to the chopsticks at each philosopher's left and right sides. Additional parameters provide the desired number of bites to eat and the identification number of the action.

The action code in Figure 3.12 begins with the now familiar "boilerplate" aliasing of the argument pointers and setting of triggers on the chopsticks. It is unnecessary to set a trigger on the `bite_count` since we know that the chopsticks change state on every significant occasion and will provide adequate triggering.

The logic of the philosopher action code is quite straightforward. It gets the chopsticks (if apparently free), then immediately dines and releases them. After each bite it checks to see whether it has finished the meal. The crucial aspect of this program is that the actions acquire the chopsticks by first testing the ownership field, and, if both are free, writing their own ID to the fields of the two distinct data objects. Affinity guarantees that the effect of such operations within an action will be atomic. Not only will at most one action succeed in writing to each individual chopstick object, but at most one action will succeed in writing to the pair of chopstick objects required to dine. Such mutual exclusion, in modification of individual objects, and consistency, in modification of sets of objects, derives from the write-set coherence guarantee of Section 2.6.3. The issues of deadlock that arise from solutions that require explicit sequential locking of objects are removed from the programmer's concern by action atomicity in the Affinity

```
int next(int i) { return (i+1) % N ; }

root()
{
  int           i;
  chopstick_p   chop_dict[N];
  int           *bites_left;
  printf("%d dining philosophers want %d bites\n",N,NBITES);
  for (i = 0; i < N; i++)
    chop_dict[i] = new chopstick;
  for (i = 0; i < N; i++) {
    *(bites_left = new int) = NBITES;
    SHOW("Making philosopher %d with a %d bite appetite\n",
         i,*bites_left);
    spawn(philosopher(chop_dict[i],chop_dict[next(i)],bites_left,i));
  }
}
```

Figure 3.11: Dining Philosophers solution **chop** — root action

computational model.

The action's view of the state of the chopsticks may or may not reflect the actual current values. If one of the chopsticks appears to be in use, the action will do nothing but exit. When the chopstick is freed, the action will be triggered to reevaluate the situation. If the action incorrectly views a chopstick as free when it is in fact in use, the action will fail in its effort to write to the owner field when the kernel discovers that the modified object was not the current version (see Section 2.5.1). The implementation supporting this feature is discussed in section 4.8.

3.6 A Bounded Buffer

Figures 3.13 and 3.14 show a simple implementation of a bounded buffer, a standard concurrent programming example [6]. This buffer is a first-in first-out (FIFO) data structure containing at most a fixed number (here `NSLOTS-1`) of elements. In this example a single producer action writes (inserts) entries into it and a single consumer reads (removes) entries from it, but the code permits multiple producer and consumer actions. The finite amount of storage in the buffer imposes a requirement that the producer not write into a full buffer, inducing a form of synchronization that is more or less strict depending on the number of "slots" (or "places" or "slack") in the buffer.

Finding a solution to the producer/consumer problem is a chronic problem in message-passing operating systems, since they attempt to provide the illusion of unbounded buffering with finite storage resources. Typically it is the application programmer's responsibility to implement *ad hoc* solutions to ensure that producer processes do not

```
declare_action(
philosopher,(chopstick_p left_chop_p, chopstick_p right_chop_p,
             int_p bite_count_p, int my_id))
{
  chopstick     &left_chop = *left_chop_p;
  chopstick     &right_chop = *right_chop_p;
  int           &bite_count = *bite_count_p;
  left_chop_p.set_trigger();
  right_chop_p.set_trigger();
  if (bite_count && left_chop.unused() && right_chop.unused()) {
    left_chop.get(my_id);
    right_chop.get(my_id);
    bite_count--;
    SHOW("Philosopher %d is dining with %d bites left\n",
         my_id,bite_count);
    left_chop.release();
    right_chop.release();
    if (!bite_count) {          // All bites taken?
      left_chop_p.clear_trigger();
      right_chop_p.clear_trigger();
      printf("Philosopher %d is done eating\n",my_id);
    }
  }
}
```

Figure 3.12: Dining Philosophers solution **chop** — action declaration

saturate the system's buffering capacity; failure to do so may be catastrophic. The virtually-shared-memory programming model of Affinity allows the current capacity of a specific buffer to be readily examined; in a message-passing system comparable information is not available from the system and may be impossible to generate without dramatic complication of the user program. The accessibility of this information, coupled with the elegance of action abortion as a rescheduling mechanism, demonstrate some particular advantages of Affinity over a pure message-passing system.

The example code given here buffers integers in an array. More complex data elements can be buffered, e.g., structures or references to other data blocks, so this buffering technique (and its elaboration in Section 3.7) is quite general.

Figure 3.15 is a program **prodcon1** using the `buffer` class, Figure 3.16 shows the output of **prodcon1**.

3.6.1 Data Objects and Data Blocks

The `buffer` class declared in Figure 3.14 (as a `struct`, the unprotected alternate keyword) is the first data object with a nontrivial structure. The C++ `struct` contains

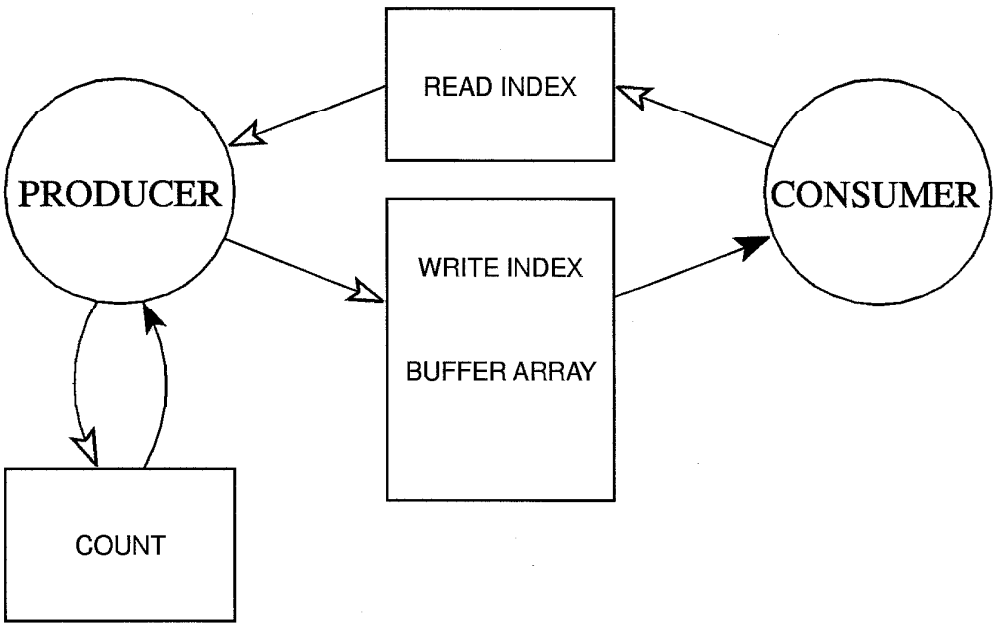

Figure 3.13: `buffer` object in **prodcon1** program

three *member variables*: an array used as a circular buffer for actual data storage and two indices, recording the current write and read points in the buffer.

Given that a C++ subset is the standard programming language for Affinity, it is natural to think of data blocks as *objects*, instances of C++ classes, but a pair of conceptual distinctions must be remembered. First, only instances resulting from an explicit allocation, usually by the C++ `new` operator, are data blocks known to the Affinity runtime system. Variables automatically allocated on the stack (local function variables) are not data blocks. No global or statically allocated datum is permitted. Second, data blocks are a more primitive element than C++ objects, the more complex of which may contain several data blocks. Data blocks are virtually contiguous chunks of memory, which have a one-to-one correspondence with basic C++ types such as arithmetic types, arrays of arithmetic types, `structs`, and simple classes. Construction of an instance of a complex class may require repeated memory allocations, each of which returns a new data block. For example, the *matrix* class example defined in Section 3.8 implements a matrix as a vector of references to vectors; each vector is a distinct data block. Conversely, a single data block could contain an array of `struct` objects, so there is not a one-to-one correspondence between the two concepts. This distinction may seem pedantic to the user of a class library, but it is an essential design consideration for the programmer

```
// Producer/consumer example

#define N_WRITES        10
#define N_SLOTS         4

#include <affinity.h>
declare_pointer(int,int_p);

struct buffer {
 int   write_index;
 int_p read_index_p;
 int   b[N_SLOTS];  // buffer
       buffer()     { read_index_p = new int; }
 int   read()       { int &read_index = *read_index_p;
                      return b[read_index++ % N_SLOTS]; }
 void  write(int x) { b[write_index++ % N_SLOTS] = x; }
 int   notempty()   { int &read_index = *read_index_p;
                      return read_index < write_index; }
 int   notfull()    { int &read_index = *read_index_p;
                      return write_index - read_index < N_SLOTS; }
};
declare_pointer(buffer,buffer_p);
```

Figure 3.14: **prodcon1**, a bounded buffer implementation

implementing a new class type. The details of class implementation are properly hidden by a well-constructed interface definition, but the potential concurrency and consistency guarantees of a class are defined by the way it is constructed.

The distinction between objects and data blocks is visible in this example. The C++ compiler will automatically invoke a class *constructor* when a new instance of the class is instantiated, to allow user-defined initialization. The `buffer()` constructor in Figure 3.14 is called to create a `buffer` object, a single logical entity composed of two data blocks, one consisting of the `read_index` integer and one for all other data, including the `read_index_p` reference to the `read_index` block. The `read_index` variable is dynamically allocated whenever a new instance of a `buffer` is allocated, by the constructor

```
buffer()     { read_index_p = new int; }
```

The Affinity rules about atomicity of effect and consistency of modifications are defined in terms of data blocks, not objects, to allow concurrency of operations within a single object. While it may seem vexatious to obscure such significant mechanisms as consistency control and granularity at a fairly low level and make it implicit, it does seem fairly natural that objects allocated as a unit are modified as a unit. However,

```
declare_action(
producer,(buffer_p output,int_p count_p,int n_writes))
{
  int &count = *count_p;
  count_p.set_trigger();
  if (count < n_writes)
    if(output->notfull())
      output->write(count++);
    else
      s_abort_action();                    // Reschedule action
}

declare_action(consumer,(buffer_p input))
{
  input.set_trigger();
  while (input->notempty())
    printf("Consumed item %d",input->read());
}

root()
{
  buffer *bf = new buffer();
  spawn(producer(bf,new int,N_WRITES));
  spawn(consumer(bf));
}
```

Figure 3.15: **prodcon1**, a producer and consumer sharing a bounded buffer

failure to appreciate this programming subtlety can result in reduced concurrency on the one hand, or erroneous inconsistency on the other.

3.6.2 Code Exposition

The `root()` action instantiates a `buffer` object in the line

```
  buffer *bf = new buffer();
```

causing the `buffer()` constructor to initialize the `read_index_p` portable pointer to point to the newly allocated integer data block `read_index`, as mentioned above. One producer and one consumer action are spawned with references to the new `buffer` object. Figure 3.13 shows the `buffer` object as two data blocks, omitting the detail of the reference from the `struct buffer` to the other (`read_index`) data block. As in the previous **hello** examples, the spawn allocates an externalized counter data block (`new int`) in the producer's argument list.

```
1,711: Consumed item 0
1,711: Consumed item 1
1,711: Consumed item 2
1,711: Consumed item 3
1,711: Consumed item 4
1,711: Consumed item 5
1,711: Consumed item 6
1,711: Consumed item 7
1,711: Consumed item 8
1,711: Consumed item 9
```

Figure 3.16: Console output of **prodcon1**

The `consumer` action code is straightforward. A trigger is set on the `struct buffer` data block to cause scheduling in reaction to any write to the buffer. The line

```
while (input->notempty())
```

selects the `buffer` class method `notempty()` to check whether there is any entry in the buffer by executing the defined code

```
int   notempty()   { int &read_index = *read_index_p;
                     return read_index < write_index; }
```

which just compares the read and write pointers. The action code repeatedly calls the `read()` method, printing the resulting integer, until the buffer is empty. Recall that the version of buffer data block visible to the action is guaranteed not to change during action execution (Section 2.6.2). It is not guaranteed that the `struct` is the current version; it doesn't matter, since a write to the buffer that occurs after the consumer action has begun execution will subsequently trigger another activation of the consumer action. The significance of input staleness is lessened by the triggering mechanism for reactive scheduling, (see Section 6.3.3). Consistency of the write index and the buffer-array content are important, therefore, they are placed together in one block. The `read` method modifies the `read_index` data block, which guarantees that the consumer action operates on the current version of that variable (Section 2.6.3).

The producer action code is slightly more complicated.

3.6.3 Action abortion

The essential difficulty of the bounded buffer producer/consumer problem is that production must somehow be restricted when the buffer is full. In this example, the producer action simply aborts its own execution, forcing its own failure by explicitly calling a system service in the line

```
s_abort_action();                    // Reschedule action
```

when the buffer is found to be full. An action that fails, whether from an explicit call to the `s_abort_action` service or due to an implicit object access conflict, has no visible effect on the system state as recorded in the content of data objects in the user's computation. (Action failure can be inferred by examining kernel data objects that record performance data, including action failures, but this information is not germane to user-program logic.)

Note that Affinity semantics do not allow the producer code to busy-wait, looping while testing for a full output buffer to clear: it is guaranteed not to reflect external changes during action activation. The action would loop endlessly, a programming error.

A legitimate coding alternative for the

```
s_abort_action();                    // Reschedule action
```

statement would be

```
touch(count_p);                      // Reschedule action
```

which is a predefined function that performs a non-destructive write to the argument address, causing the object to be marked as modified. This would cause the producer action to be rescheduled, since it has set a trigger on the count integer object. In this simple example there is no reason to prefer one alternative over the other. In more complicated code aborting the action may simplify the programming if the action has modified other objects before some untoward condition is detected.

3.6.4 Non-Conflict of Producer and Consumer

The producer action may fail if the buffer becomes full, and it is forced to abort and be rescheduled. The consumer action does not fail. It is worth noting that there is no access conflict between the producer and the consumer actions, since the producer modifies only the block containing the write index and the buffer array `b`, while the consumer modifies only the `read_index` data block. No actions need fail in order to maintain data block coherence, since each block has a single writer.

3.6.5 Multiple Producers and Consumers

The **prodcon1** program (Figure 3.15) has a single producer and consumer action, but the `buffer` object class definition (Figure 3.14) can be used without change for multiple producers and consumers. Figure 3.17 shows a `root()` action for **prodcon2** that spawns multiple producer and consumer actions, defined as before, all operating on one `buffer` instance. Contrary to the previous example, there is a possible conflict if multiple consumers attempt to modify the read index block concurrently: only one will succeed in doing so. The situation is the same for the other block that holds the write index. Producer actions contend with other producer actions, consumers with consumers. Producers do not conflict with consumers.

Figure 3.18 shows the output from a single-producer, four-consumer case. Note that the output is ordered for each multicomputer node, but is not totally ordered. In fact, the stored values were read from the global buffer in order, but multiple consumer actions concurrently printing to the console induce some nondeterminacy. The important points are that each value is read by exactly one of the consumer actions, and that there is no inconsistency in the effect of the multiple consumer actions.

```
#define N_PRODUCERS     1
#define N_CONSUMERS     4

root()
{
  int i;
  buffer *bf = new buffer();
  for (i = 0; i < N_PRODUCERS; i++)
    spawn(producer(bf,new int,N_WRITES));
  for (i = 0; i < N_CONSUMERS; i++)
    spawn(consumer(bf));
}
```

Figure 3.17: **prodcon2**, multiple producers and consumers sharing a bounded buffer

```
4,711: Consumed item 0
2,711: Consumed item 1
4,711: Consumed item 4
2,711: Consumed item 2
1,711: Consumed item 6
4,711: Consumed item 5
2,711: Consumed item 3
1,711: Consumed item 7
1,711: Consumed item 8
1,711: Consumed item 9
```

Figure 3.18: Console output of **prodcon2**, 4 consumers

3.7 A Symmetric Buffer

The producer/consumer example of Section 3.6 demonstrated how a C++ object could be implemented as two Affinity data blocks to allow non-conflicting (failure-free) access to a unidirectional bounded buffer. Figures 3.19, 3.20 and 3.21 show a completely symmetric elaboration of the buffer scheme previously introduced. The `sbuffer` class is also implemented as two data blocks, with the same non-conflicting-access properties as the `buffer` class, but providing bidirectional bounded buffering. The paired blocks contain references to each other; their usage is distinguished only by which "end" of the bipartite object is passed to the "trader" actions.

The diagram of the symmetric buffer, Figure 3.19, should be compared with that of the asymmetric producer/consumer buffer of Figure 3.13. The actions no longer have distinct character or code. The triggered flow of control caused by `sbuffer` writes is symmetric and cyclic. A possible "chicken-and-egg" conundrum is resolved by the

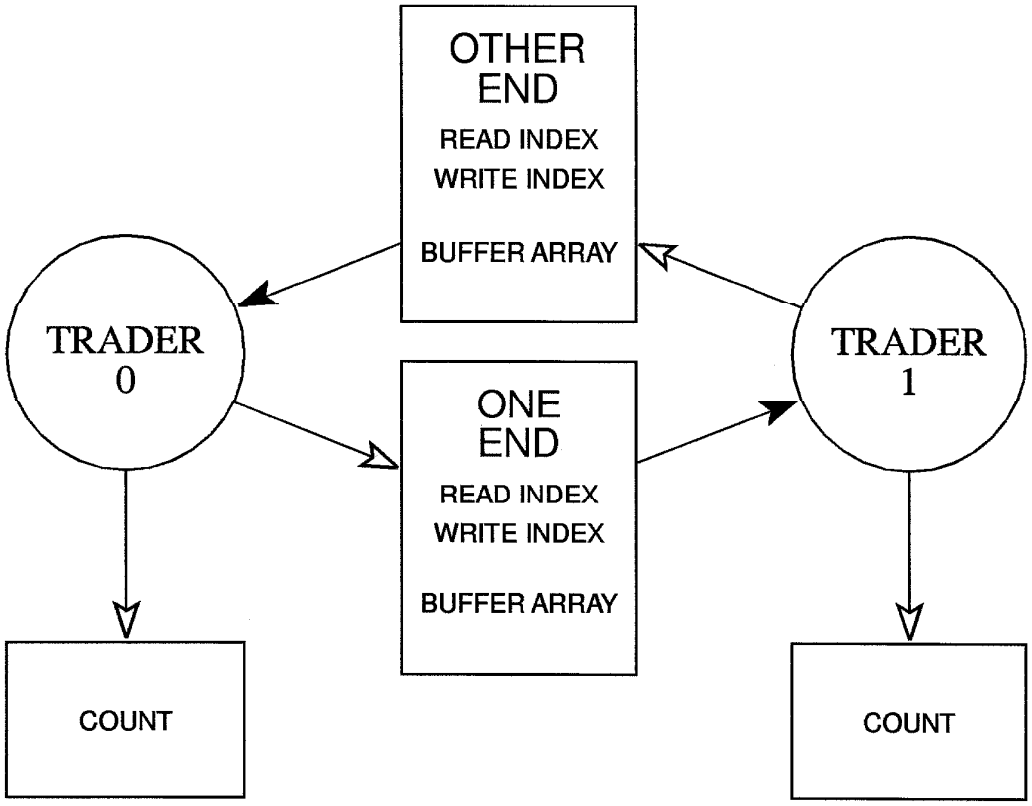

Figure 3.19: `sbuffer` object in the symmetric buffer example. The read indices refer to the other data block buffer. An action is triggered by either a read or write by the other action, unlike the producer/consumer example which was asymmetrically driven by the counter.

fact that Affinity actions "come up running" at instantiation. The first activation of each "trader" action causes a write into an empty output buffer, driving subsequent computation. The "count" objects are checked to determine whether each "trader" action has written the desired number of items into its output buffer, but activation is triggered by either a write to its input buffer or a read from its output buffer. Since the `read_index` of the output buffer is in the same data block as the input buffer and its index, either event modifies the same data block. The fact that the triggering cause is ambiguous causes no particular difficulty because it is easy to examine the state of multiple possible event sources.

```
// Symmetric buffer example

#define N_ITEMS0        3
#define N_ITEMS1        5
#define N_SLOTS         2

#include <affinity.h>
declare_pointer(int,int_p);

struct sbuffer {
  int     read_index;  // refers to other end's buffer
  int     write_index; // refers to this end's buffer
  ref     other_end_p; // untyped portable pointer to other end
  int     b[N_SLOTS];  // buffer
          sbuffer()    { other_end_p = new sbuffer(this); }
          sbuffer(sbuffer *oe)
                       { other_end_p = oe; }
  int     read()       { int &read_index = other_end()->read_index;
                         return b[read_index++ % N_SLOTS]; }
  void    write(int x) { b[write_index++ % N_SLOTS] = x; }
  int     notempty()   { int &read_index = other_end()->read_index;
                         return read_index < write_index; }
  int     notfull()    { int &read_index = other_end()->read_index;
                         return write_index-read_index < N_SLOTS; }
  sbuffer *other_end() { return other_end_p; }
};
declare_pointer(sbuffer,sbuffer_p);
```

Figure 3.20: **sbuffer**, a symmetric bounded buffer implementation

3.7.1 Code Exposition

The `sbuffer` class (Figure 3.20) is very similar to the `buffer` class (Figure 3.14) of the producer/consumer example. The main difference lies in the two data blocks being now identical; previously, the second was just a dynamically allocated integer holding the `read_index`. Instead of an `int_p` portable pointer to `int`, we have the line

```
  ref     other_end_p; // untyped portable pointer to other end
```

The predeclared system type `ref` is the "untyped" base class of Affinity portable pointers. Its use here simplifies a forward declaration in the code; it is used to contain an `sbuffer_p` portable pointer. The `other_end()` method is an access function that returns a version of the untyped `ref` that is automatically coerced to the proper type.

C++ allows functions with distinct parametric "signatures" to use the same name, as is seen in the two versions of the `sbuffer` constructor. The `sbuffer()` constructor, with

```
declare_action(trader,(sbuffer_p output,int_p count_p,int n_items))
{
  int       &count = *count_p;
  sbuffer_p input = output->other_end();
  input.set_trigger();
                                          // Producer aspect
  while ((count < n_items) && (output->notfull())) {
    output->write(count++);
    if (count == n_items)
      printf("Production done");
  }
                                          // Consumer aspect
  while (input->notempty())
    printf("Consumed item %d",input->read());
}

root()
{
  sbuffer *one_end = new sbuffer();
  printf("Traders exchanging %d and %d items, buffer has %d slots\n",
          N_ITEMS0,N_ITEMS1,N_SLOTS);
  spawn(trader(one_end,new int,N_ITEMS0));              // Trader 0
  spawn(trader(one_end->other_end(),new int,N_ITEMS1)); // Trader 1
}
```

Figure 3.21: **symmbuffer**, a symmetric bounded buffer program

no arguments, is the one called by the user code. As in the `buffer` class example, the constructor allocates a second data block. However the `buffer()` constructor simply allocated an `int`. The `sbuffer()` constructor must allocate another `sbuffer`, but this could cause an infinitely recursive sequence of further `sbuffer` allocations. To avoid this error, the default constructor `sbuffer()` calls the `sbuffer(sbuffer *)` variant constructor with argument `this`. The C++ `this` keyword is similar to `self` in Smalltalk: it is a pointer to the current instance of a class, usually implicit, but here required to stitch together the forward and backward links of the data block pair.

The "trader" action definition of Figure 3.21 is a slightly simplified concatenation of the "producer" and "consumer" actions of Figure 3.15. Instead of two types of actions with distinct behavior, there is one action type with two behavioral aspects. Depending on the evaluation of the logical "guard" expressions in the `while` statements, the "trader" action may act as either or both a producer and consumer. The initial activation will cause the producer code to write items to the output buffer, either completing that task or filling the buffer. In either case the action will be triggered after the other action reads the buffer, and will write again as needed. As long as one

```
31,711: Traders exchanging 3 and 5 items, buffer has 2 slots
0,711: Consumed item 0
1,711: Consumed item 0
0,711: Consumed item 1
1,711: Consumed item 1
0,711: Production done
1,711: Consumed item 2
0,711: Consumed item 2
1,711: Production done
0,711: Consumed item 3
0,711: Consumed item 4
```

Figure 3.22: Console output of **symmbuffer**

action continues to write to its output buffer, the other will be triggered to read its input buffer (the same data block). Because the "trader" actions are driven by their interactions with the `sbuffer` object instead of the counter, it is not necessary to have an action-abort option in the code. Because of the atomicity of effect of Affinity actions, the ordering of the producer- and consumer-aspect code is unimportant.

Figure 3.22 shows the console output of the program. The example has differing numbers of items exported by each "trader" to make the point that whereas the code is symmetric, the actual usage is not constrained to symmetric exchanges. If we were to change the constant definitions to read:

```
#define N_ITEMS0        8
#define N_ITEMS1        0
```

we would have a producer/consumer case.

3.7.2 Demand- and Supply-Driven Actions

In most message-passing environments, programs tend to be "supply-driven:" processes react to and process messages received from an unsolicited sender. Since received messages can be queued in receiving nodes, this can reduce the effect of communications latency, at the cost of increased storage requirements in the receiver. Message-passing programs can be written in a "demand-driven" style with requestor messages, but the increased program complexity and exposure to unwanted synchronization and latency costs are deterrent. RPC-based programming systems tend to be "demand-driven," with data objects or servers replying to a request; some simpler message-passing systems that support only a synchronous message-exchange rendezvous style are effectively demand-driven. Buffering requirements are reduced, but concurrency may suffer and, in synchronous programming styles, load balance and data distribution become recurrent preoccupations.

The symmetric buffer example is a particularly flexible framework for blending styles of communication between Affinity actions. The producer aspect of the "trader" action

is effectively demand-driven, the consumer aspect supply-driven. An actual application's action code could make one aspect vestigal or absent. The tightness of synchronization between actions is affected by the number of places in the two unidirectional buffers chosen; these sizes could be varied dynamically in a more sophisticated implementation. CSP-style [24] unbuffered channels (zero-place buffers) cannot be emulated in Affinity; such a degree of synchronization is considered harmful. The configurative flexibility of symmetric buffers will be used later (Section 6.3.2) to simplify a discussion of program structuring techniques.

3.8 A Matrix Multiplication Example

Matrix multiplication provides an opportunity to show how a simple and familiar algorithm can be expressed concurrently in the Affinity programming model. This example also demonstrates an implementation of two common data structures, and introduces the use of an observational termination-detection implementation.

The basic program design decision in Affinity is how to structure the data as data objects. The manner in which the data is divided into distinct objects defines the granularity of the program, and, to a great extent, its potential concurrency. A rudimentary matrix-arithmetic library is used in this and the following program, declaring `vector` and (square) `matrix` classes as shown in Figure 3.24. The C++ operator-overloading facility is used extensively to minimize extraneous detail in the program code proper.

The key point to observe in this matrix implementation is that the `matrix` object is composed of `vector` objects, each representing one row or column. This structure is evident in Figure 3.23, where the references to the row or column vectors in the "matrix" vectors are represented by the ball-ended lines. The `matrix()` constructor in Figure 3.24 creates a `matrix` object, a logical entity composed of several data blocks, one for the `matrix struct` and one for each `vector struct`. (The `matrix()` constructor is declared in the `matrix` class declaration, but defined separately. The `::` notation is a C++ scoping mechanism to indicate the relevant class.) If the matrix were a single monolithic entity, the potential for concurrent operations would be very limited due to Affinity atomicity rules. In this implementation, each vector may be modified independently and concurrently. Figure 3.23 shows just one action operating on a row of the input and output matrices; there is one such action per row of the matrix.

3.8.1 Matrix Multiply Code Exposition

Figures 3.24 and 3.25 show excerpts from the included file ``` ``matrix.h'' ```. The struct `vector` contains a fixed size array of (previously `#defined`) `elem_types`, (e.g., `int` or `float`). Two overloaded binary operators are declared, a standard array indexing operator `[]` and a dot product operator `^`. As noted above, the struct `matrix` contains a fixed size array of `vector_ps`, portable pointers to `vector` objects. Two overloaded binary operators are declared, a standard array indexing operator `[]` and a multiplication operator `*` (not to be confused with the unary dereferencing operator `*`), defined in Figure 3.27. The dot product `^` definition in Figure 3.25 is unremarkable standard C++ but the `transpose` method merits comment. The `matrix` class partitions the matrix by rows, but, for the matrix multiplication `C = A * B`, it is convenient to access `B` by

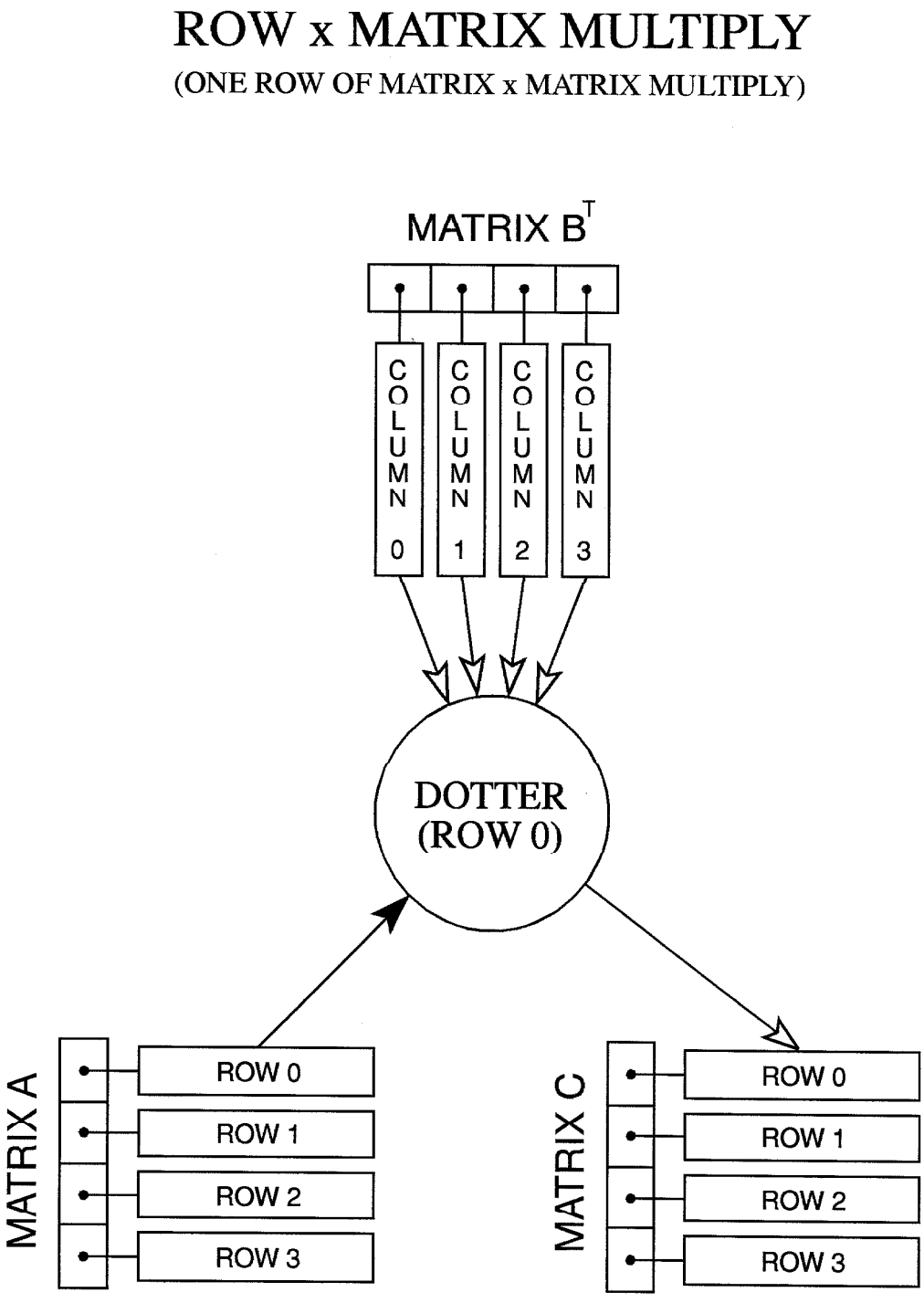

Figure 3.23: Matrix multiply program

```
struct  vector  {
  elem_type     v[ORDER];
  elem_type     operator^ (vector &a);  // Dot product operator
  elem_type &   operator[] (int i)      { return (v[i]); }
  void          print();
};
declare_pointer(vector,vector_p);

struct  matrix  {
  vector_p      m[ORDER];
                matrix();
  vector &      operator[] (int i)      { return *m[i]; }
  matrix *      operator*(matrix &b);
  matrix *      transpose();
  void          print();
};
declare_pointer(matrix,matrix_p);

matrix::matrix()
{
  int   i;
  for (i = 0; i < ORDER; i++)
    m[i] = new vector;
}

void
matrix::print()
{
  int   i;
  for (i = 0; i < ORDER; i++)
    m[i]->print();
}
```

Figure 3.24: Matrix arithmetic class definition

columns. This program simply produces a new matrix, B transpose, which is then accessed by rows. This choice makes for simple code and also reflects that several sorts of inefficiency attend non-sequential access of memory. For brevity, this potentially concurrent operation is performed sequentially since it is relatively quick. Efficiency is also improved in this case by using local (stack) copies of the portable pointers `m[i]` in the matrices, because there is some overhead in dereferencing the portable pointers, and because the required indexing pattern is at odds with the loop nesting. (The details of this issue and related optimizations are discussed in Section 4.11.)

```
elem_type
vector::operator^(vector &a)
{
  int   i;
  elem_type     sum = 0;
  for (i = 0; i < ORDER; i++)
    sum += v[i] * a.v[i];
  return(sum);
}

matrix *
matrix::transpose()
{
  matrix        *t = new matrix;
  vector        *a[ORDER],*b[ORDER],*v;
  int           i,j;
  for (i = 0; i < ORDER; i++) {
    a[i] = m[i];        // Dereference vectors for efficiency
    b[i] = t->m[i];
  }
  for (i = 0; i < ORDER; i++) {
    v = b[i];
    for (j = 0; j < ORDER; j++)
      (*v)[j] = (*a[j])[i];
  }
  return(t);
}
```

Figure 3.25: Matrix arithmetic class definition

An examination of the `root()` function in Figure 3.28 shows that the introduction of concurrency into the program is implicit in the line

```
cp = a*b;
```

which invokes the overloaded operator * defined in Figure 3.26. A "dotter" action (Figure 3.27) is spawned to compute the dot products for each element of a row of the result matrix `C`; the arguments passed are pointers to a row vector of `A`, matrix `B` transpose, and a row vector of the output matrix `C`, respectively.

3.8.2 Initialization Code

A new system service is introduced in Figure 3.27 in the line

```
if (s_initial())
```

```
// Matrix multiply example

#define ORDER           128
#define N_ITERS         10
#define elem_type       float

#include <affinity.h>
#include "checkTermination.h"
#include "matrix.h"
declare_pointer(int,int_p);
```

Figure 3.26: Matrix multiplication

`s_initial()` returns a true value only for the first activation of a given instance of a spawned action. In this case it serves only to improve efficiency by preventing redundant attempts to set triggers. More substantively, it allows any desired initialization code to be run, e.g., printing a startup message or performing member data initialization. Use of this mechanism to initialize a data object is logically redundant, since C++ constructors can do so, but can be practically valuable. Constructor initialization occurs when the object is dynamically allocated; often a parent action will allocate many objects and pass these references to child actions it spawns. Constructor initialization by the parent is therefore sequential; in contrast, explicit initialization code in the child actions may be executed concurrently. (Of course, constructors could spawn initializing actions to gain concurrency, but the necessity to check for completion of initialization adds complication.)

3.8.3 Spawn Variants

The `root()` action (Figure 3.28) contains a variant of the `spawn` macro

```
spawnn(print_results(ap,cp,ct,s_ticks(),N_ITERS,new int),
                    "=Print results");
```

which takes two arguments: the action with its argument list and a string containing a name identifying this instance of the action to the kernel. The instance name can be used for the purposes of debugging and performance measurement; additionally, the action-instance name provides a limited means by which the programmer can influence the placement of actions on physical multicomputer nodes. By convention, an action with an instance name beginning with "=" ("equal") will be be loaded onto the same node as the current action; an action with an instance name beginning with "_" ("underscore") will be not be relocated from whichever node on which it is initially loaded. The motivation for this very limited control is to improve the accuracy of timing measurements by allowing actions to use the system clock on the same node, since the node clocks are poorly synchronized with each other. The system service `s_ticks()` returns the value of a 100Hz node clock.

```
declare_action(
dotter,(vector_p arowp, matrix_p btp, vector_p crowp))
// Computes one row of the matrix product
{
  matrix     &bt = *btp;
  vector     &arow = *arowp, &crow = *crowp;
  int        i;
  if (s_initial())
    arowp.set_trigger();              // React to A changes only
  for (i = 0; i < ORDER; i++)
    crow[i] = arow ^ bt[i];
}

matrix *
matrix::operator*(matrix &b)
{
  unsigned   start_time = s_ticks();
  matrix     *btp = b.transpose();       // B transpose matrix
  matrix     *cp = new matrix, &c = *cp; // Product matrix
  int        i;
  for (i = 0; i < ORDER; i++)
    spawn(dotter(m[i],btp,cp->m[i]));
  printf("Multiplication setup time = %d centiseconds\n",
         s_ticks()-start_time);
  return (cp);
}
```

Figure 3.27: Matrix multiplication

3.8.4 Termination Detection

As coded, the program requires information from the kernel to detect termination, i.e., the completion of all the scheduled "dotter" actions that compute the content of product matrix. The line

```
checkTermination      *ct = new checkTermination;
```

in the `root` function instantiates a library object that indicates the termination status of the user computation. The "print_results" action is triggered when this object changes state. The object checks for termination with the line

```
if (terminated->done())
```

For the purpose of performance analysis the program repeats the matrix multiplication `N_ITERS` times. The "dotter" actions set triggers on the row vectors of matrix `A` and recompute the `C` matrix whenever A is disturbed. The "print_results" action in

```
declare_action(
print_results,(matrix_p a_p,matrix_p c_p,
               checkTermination_p terminated,
               unsigned starting_time, int n_iters,
               int_p iter_count_p))
{
  matrix        &a = *a_p, &c = *c_p;
  int           &iter_count = *iter_count_p;
  if (s_initial())
    terminated.set_trigger();
  if (terminated->done()) {
    printf("%d matrix multiplication(s) took %d centiseconds\n",
           ++iter_count,s_ticks() - starting_time);
    c.print();
    if (iter_count < n_iters)
      for (int i = 0; i < ORDER; i++)
        touch(&a[i]);
  }
}

root()
{
  int                   i;
  matrix                *ap = new matrix, &a = *ap;
  matrix                *bp = new matrix, &b = *bp;
  matrix                *cp;
  checkTermination      *ct = new checkTermination;
  printf("Matrix multiply example, %dX%d\n",ORDER,ORDER);
  for (i = 0; i < ORDER; i++) {
    a[i][i] = 10; b[0][i] = 30; b[i][i] = 20;
  }
  cp = a*b;     // Use of pointer simplifies class definition
  spawnn(print_results(ap,cp,ct,s_ticks(),N_ITERS,new int),
                       "=Print results");
}
```

Figure 3.28: Matrix multiplication

Figure 3.28 contains a loop that nondestructively writes to each row of `A` to cause the "dotter" actions to recompute the multiplication:

```
    if (iter_count < n_iters)
      for (int i = 0; i < ORDER; i++)
        touch(&a[i]);
```

It is a noteworthy curiosity that a "one-shot" concurrent program that does not depend on the scheduling mechanism of Affinity triggers can be converted to a sequential program by interpreting the **spawn** macro as a function call. This particular example can be executed one time as a sequential program in this way, simply by a macro definition of the word `SEQUENTIAL`, which alters the **spawn** macro. This property is useful in providing accurate performance comparisons of the sequential and concurrent versions of the program (see Section 5.1.2).

3.9 An All-Points Shortest Path Algorithm

The all-points shortest path (APSP) graph problem [2] is particularly well-suited to the Affinity programming model. Given a set of vertices and connecting edges with an associated cost, the APSP problem requires the computation of the square matrix of minimal-cost paths between all vertices. The Floyd-Warshall algorithm [2] is inefficient for sparsely-connected graphs and unsuitable for asynchronous concurrent execution. Dijkstra's single-point shortest path (SPSP) algorithm [2] can be executed concurrently for each vertex; however, an efficient version of the algorithm requires substantial code to implement a priority queue. This approach also replicates all the graph data structures on each processing node. While the basic step of all APSP programs is simple, the computation of a triangle inequality, the above sequential algorithms devote a great deal of time (Floyd-Warshall) or code (Dijkstra) to ensuring that the steps are computed in the proper order. The Affinity model allows this sort of scheduling to be done by the kernel instead of the user code.

A simple and intuitive concurrent algorithm with performance comparable to Dijkstra's algorithm for low-diameter graphs is presented below. For a naturally concurrent problem such as APSP, a concurrent solution seems more natural than independent, parallel execution of a relatively complex sequential SPSP algorithm.

3.9.1 APSP Program Code Exposition

Three methods (Figure 3.29) are added to the class `vector` to improve the conciseness of the APSP program. The constructor

```
vector::vector (int x)
```

initializes all vector elements to a given value. The "delta assignment" operator `<<=` copies each vector element from the input vector to the output vector if and only if the corresponding elements differ. If all elements differ, it is identical to a standard assignment operator `=`; if all the elements of the input and output vectors are the same, it is a "no-op." This is simply a convenient way to allow fixed-point programs to terminate by ceasing to modify output objects. The `min()` method is rather specialized to the APSP problem: it computes a triangle inequality and keeps a running minimum in the vector.

This particular example program generates a test graph (in `root()` Figure 3.33) of a binary n-cube (hypercube), but the program is general for sparse graphs. Only the choice of an array (for simplicity) to store the list of incident edges to a vertex (in the `edge_list` class in Figure 3.32) is a practical restriction on the input graphs, restricting graph degree to `MAX_DEGREE`.

```
vector::vector (int x)
{
  int i;
  for (i = 0; i < ORDER; i++)
       v[i] = x;
}

vector &
vector::operator<<= (vector &s)
{
  int i;
  for (i = 0; i < ORDER; i++)
    if (v[i] != s[i])
      v[i] = s[i];
  return *this;
}

vector &
vector::min (vector &s,int x)
{
  int i;
  for (i = 0; i < ORDER; i++)
    if (v[i] > s[i] + x)
      v[i] = s[i] + x;
  return *this;
}
```

Figure 3.29: Vector class definition

The "vertex" action, sketched in Figure 3.30 with code listed in Figure 3.32, is the heart of the APSP program. Each graph vertex has an associated "vertex" action and a vector (`*current_costs`) that records the cost of the current minimum path from all vertices to this vertex ("self_vertex"). The action computes the minimum of path costs over all the incident edges to the vertex. Triggers are set on the cost vectors of each neighboring vertex (i.e., at the other end of an incident edge), causing a reevaluation of the minimum path cost to "self" whenever a neighbor's cost vector changes. A `tmp` vector is instantiated on the stack and initialized to a number no smaller than the graph diameter. The statement

```
tmp.min(*in_list[j].neighbor,in_list[j].cost);
```

causes `tmp` to be a running (element-wise) minimum vector over the incident edges examined. When all edges have been checked `tmp` contains the minimum path costs based on the current values of the neighbor vertex cost vectors. If the current cost vector differs from the computed minimum the "delta assignment" operation

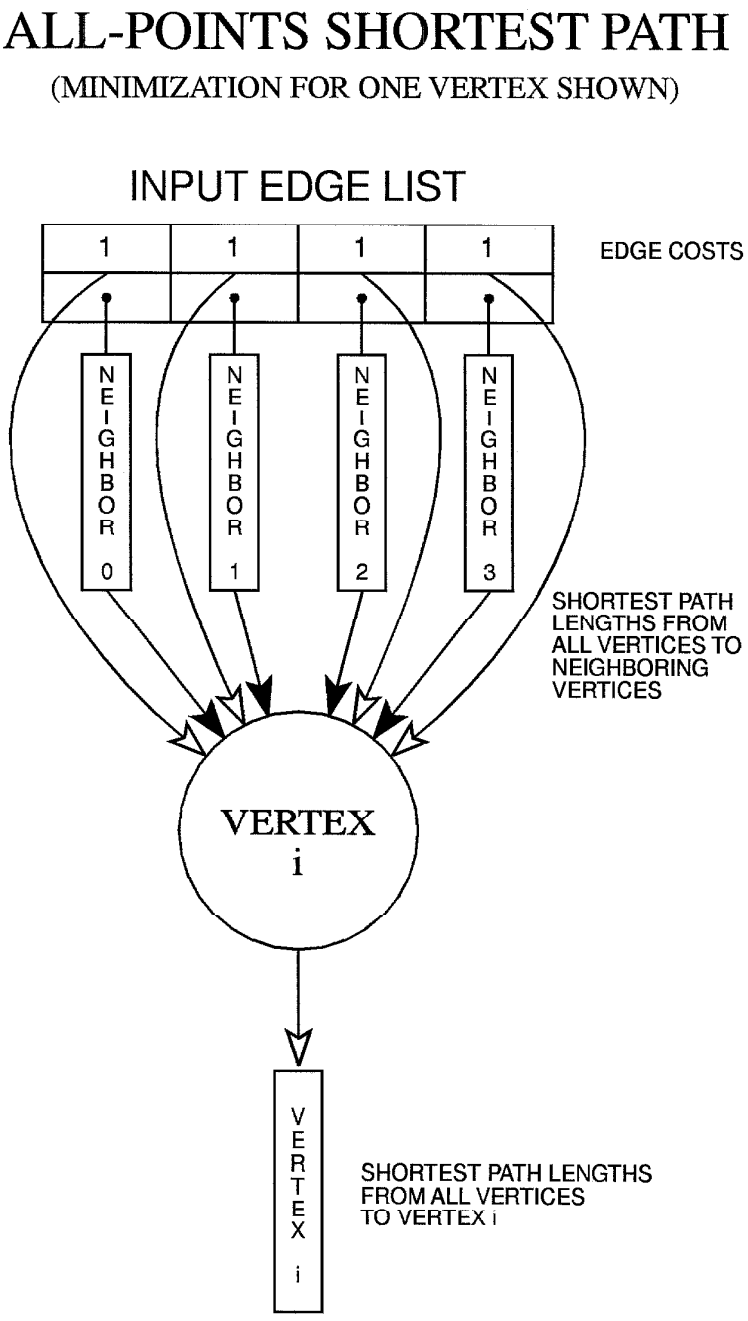

Figure 3.30: All-Points Shortest Path program, single action per vertex. A change in any of the "neighbor" cost vectors triggers a new minimum calculation for all incident edges. There is a single writer to each cost vector.

```
#define dimen           6
#define n_vertices      (1 << dimen)
#define ORDER           n_vertices
#define elem_type       ubyte
#define MAX_DEGREE      10

#include <affinity.h>
#include "checkTermination.h"
#include "paths.h"
#include "matrix.h"
```

Figure 3.31: All-Points Shortest-Paths computation

```
*current_costs <<= tmp;       // Change if not already minimum
```

will modify the current cost vector to reflect the computed minimum. Note that the costs could increase if the graph should change. If the computed minimum is the same as the current value, the `*current_costs` vector is not modified and does not cause any action to be triggered. When all actions determine that they have reached a local fixed point, the computation will become quiescent and terminate globally. As in the matrix multiply example, the `checkTermination` object will then trigger the "print_result" action.

3.9.2 A Finer-Grain APSP Program

A finer-grain version of the APSP program is sketched in Figure 3.34 with the variant code in Figure 3.35. Comparison of Figure 3.34 with Figure 3.30 shows that the minimization operation, formerly performed in a single "vertex" action, is now dispersed into multiple independent "edge_min" actions. The "vertex" action is now the second level of an action spawning tree and is deleted after its initial activation.

Each "edge_min" action does comparisions of the path costs to the given vertex via a single incident edge. If a path cost is less than the current value of an element of the vertex-path-cost vector, the new lower value is written to that vector. Unlike the previous version of the program, which computed the minimum cost over all incident edges at each activation, each "edge_min" action performs a comparison only with the current cost vector and the cost calculated via its associated edge. This is a less robust method, since it can only reduce the current cost values. We could readily eliminate this quirk by letting the "vertex" action minimize per-edge partial results, at the cost of increased storage, but that would lessen this example's interest (see Section 3.9.3).

It is therefore necessary to initialize all the elements of the distance matrix to a large value in the root action. A minor variant of the matrix constructor is provided to accept an initial value, and the line of `root()` allocating the distance matrix is accordingly changed to read

```
matrix_p              dist_mat = new matrix(INFINITY);
```

```
struct in_edge  {              // Incident edge definition
  int        cost;             // Cost of this incident edge
  vector_p   neighbor;         // Ptrs to path lengths to neighbor
};

struct edge_list {             // List of incident edges to vertex
  int      degree;             // # of incident edges to vertex
  in_edge  edge[MAX_DEGREE]; // List stored as array
           edge_list(int d)      { degree = d; }
  in_edge& operator[](int i)     { return edge[i]; }
};

declare_pointer(elem_type,elem_type_p);
declare_pointer(edge_list,edge_list_p);

declare_action(
vertex,(int self_vertex,edge_list_p in_edges,vector_p current_costs))
// Find path length minimum over all in edges to this vertex
{
  int           j;
  edge_list     &in_list = *in_edges;
  if (s_initial()) {
    for (j = 0; j < in_list.degree; j++)
      // Trigger on neighbor distance change
      in_list[j].neighbor.set_trigger();
  }
  vector        tmp(INFINITY); // Temp for computing minimum
  tmp[self_vertex] = 0;        // Distance to self is zero
  for (j = 0; j < in_list.degree; j++)
                               // For each incident edge
    tmp.min(*in_list[j].neighbor,in_list[j].cost);
  *current_costs <<= tmp;      // Change if not already minimum
}
```

Figure 3.32: All-Points Shortest-Paths computation

```
declare_action(
print_results,(matrix_p dist_mat,checkTermination_p terminated,
                              unsigned starting_time))
{
  if (s_initial())
    terminated.set_trigger();
  if (terminated->done()) { // Terminated, print results
    printf("%d centiseconds equilibration time\n",
           s_ticks() - starting_time);
    printf("Distance matrix\n");
    dist_mat->print();
  }
}

root()
{
  int                   i,j,adjacent_vertex;
  edge_list_p           in_edges;
  checkTermination_p    ct = new checkTermination;
  matrix_p              dist_mat = new matrix;
  matrix                &dist = *dist_mat;
  printf("APSP for binary %d-cube graph\n",dimen);
  for (i = 0; i < n_vertices; i++) { // For each vertex
    in_edges = new edge_list(dimen);
    for (j = 0; j < dimen; j++) {   // For each n-cube dimension
      adjacent_vertex = i^(1<<j);   // Compute neighbors
      edge_list         &in_list = *in_edges;
      in_list[j].neighbor = &dist[adjacent_vertex];
      in_list[j].cost = COST;       // Uniform edge cost
    }
    spawn(vertex(i,in_edges,&dist[i]));
  }
  spawnn(print_results(dist_mat,ct,s_ticks()),
                       "=Print results");
}
```

Figure 3.33: All-Points Shortest-Paths computation

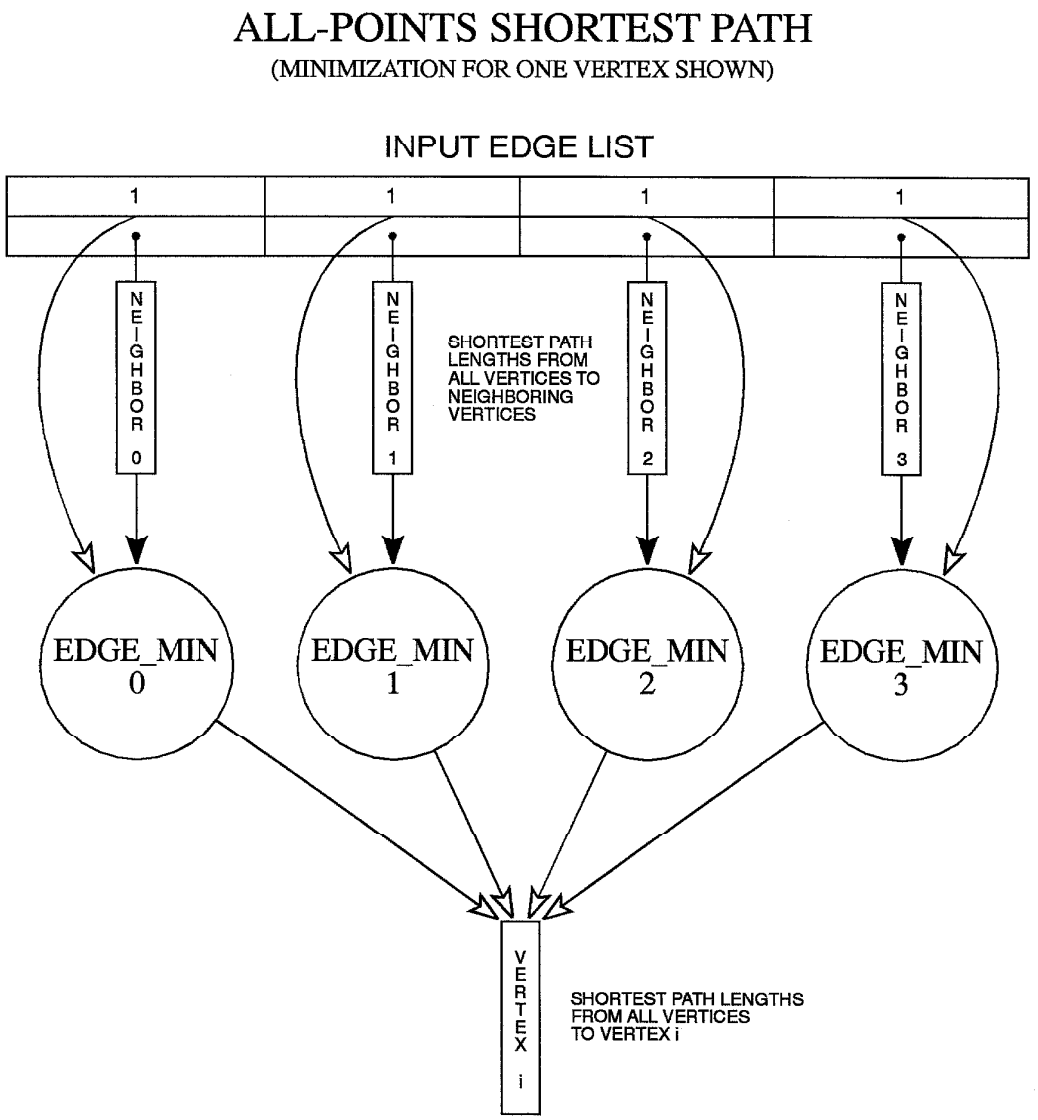

Figure 3.34: Finer-Grain All-Points Shortest Path program, one action per incident edge, multiple actions per vertex. A change in a "neighbor" cost vectors triggers a new minimum calculation for only one incident edge. There are multiple writers to each cost vector, potentially causing action failure due to implicit mutual exclusion.

```
declare_action(edge_min,(int_p edge_cost,vector_p neighbor_costs,
               vector_p current_costs))
// Find the minimum of current path and path via one edge
{
  if (s_initial())
    neighbor_costs.set_trigger();
    // Trigger on neighbor distance change
  current_costs->min(*neighbor_costs,*edge_cost);
}

declare_action(
vertex,(int self_vertex,edge_list_p in_edges,vector_p current_costs))
// Spawn minimizer actions for each edge incident to this vertex
{
  int           j;
  edge_list     &in_list = *in_edges;
  vector        &my_costs = *current_costs;
  my_costs[self_vertex] = 0;           // Distance to self is zero
  for (j = 0; j < in_list.degree; j++) // For each incident edge
    spawn(edge_min(&in_list[j].cost,in_list[j].neighbor,
                   current_costs));
}
```

Figure 3.35: Finer-grain all-points shortest-paths computation

(This is acceptable but unnecessary in the original program.) The rest of the previous version's code remains unchanged.

The ease with which the minimization code can be fundamentally restructured exemplifies the notion that composition of reactive objects can be more straightforward than for programs with less sharply demarcated interfaces and explicit control flow.

3.9.3 Sequentialization by Mutual Exclusion

In the S/2010 implementation, the performance of the finer-grain APSP program is inferior to that of the original version. The essential difference in the two versions is that in the latter the multiple writers to the `current_costs` data block will cause sequentialized evaluation of related "edge_min" actions due to implicit mutual exclusion and action failure. The first version, with single writers only, is failure-free. Also, the finer-grain actions can amortize the system's activation overhead of the action over fewer computations. For the program shown, the minimization over all incident edges performed by the first version is generally worthwhile, since path improvement is occurring concurrently on most of the incident edges on each step. In a less richly connected graph or for an incremental, localized change, the minimization over all edges could be less effective and the more precise control of the finer-grain version might prove more

efficient.

3.9.4 Reactive Scheduling

Fixed-point or relaxation problems such as APSP are naturally expressed in terms of working to establish relations between data objects, which are readily expressed with Affinity reactive actions. The APSP program can benefit from the reactive scheduling properties of Affinity triggers in another way apart from code simplification. The APSP program runs until it has computed the minimal-cost paths for the given graph, then becomes quiescent.

In the interest of brevity, this program does not set triggers on the edge costs from neighboring vertices, but it is trivial to do so. With these triggers set, any change in the graph edge weights will cause the APSP computation to resume until the path costs are again correct. Only as much recalculation will be done as is required to compute and confirm the new result; unaffected portions of the graph will not be recomputed. If the graph should represent network communications costs, for example, failures of node links or variations in cost due to loading would cause automatic recomputation of routing costs, a desirable feature achieved efficiently without any special programming effort.

Another interesting feature of the system is that a nonterminating computation, for example, a load-balancing program, can provide useful results provided that its component individual actions terminate. An ideal solution may be neither achievable nor necessary in many practical optimization problems; a good approximation may be adequate. Since the state of a continuing optimization calculation is visible to other components of a program, intermediate results from a non-terminating program may be useful. No special access method is required to communicate such results.

Chapter 4

Implementation Issues

Affinity has been implemented on the Ametek S/2010, a second-generation multicomputer with dramatically improved communications capabilities compared to previous multicomputers [42, 43, 44]. The decision to implement a special-purpose kernel was motivated by the desire to exploit novel capabilities of the S/2010 hardware, and was abetted by the author's familiarity with the S/2010. The Affinity programming model, of course, can be supported in other environments. This chapter discusses some of the goals and design decisions of Affinity and specific details of the S/2010 kernel implementation.

4.1 Design Considerations

The primary design goal of the Affinity programming model was to create a convenient expressive notation requiring a small set of primitive operations that can be executed efficiently with current multicomputer technologies.

We reiterate some of the goals for the expressive level.

Abstraction from Hardware Detail - particularly the elimination of explicit hardware references in addressing and resource allocation.

Ease of Data Sharing - including low-level support for shared data and mutual exclusion to encourage medium granularity in coding,

Implicit Communications and Coherence Properties - to remove the task of specifying computations from that of maintenance of the environment of those computations.

Reactive Scheduling - to eliminate the specification of large-scale control flow and to encourage relational and structural reasoning in program design.

Implementation goals are less abstract.

Simplicity - which never requires justification.

Scalability - to highly concurrent hardware, by choosing decentralized, distributed, mechanisms.

Efficiency of Execution - particularly for communication of blocked data in medium-grain computations.

Insensitivity to Latency - from both multicomputer hardware and system software sources.

Several implementation decisions are made in the service of the above goals.

Run-time Implementation - Maintaining compatibility with conventional compilers reduces the implementation difficulty and lowers the cognitive threshold for programmers familiar with existing languages. Affinity is implemented purely as a runtime system with a simplified user interface provided by a small library of C++ data types. Alternative approaches are possible, e.g., a compiler-based implementation for a processor without capable memory-management hardware.

Source Language - The facilities of C++ for language extension by means of user-defined data types allow a tidy interface for user coding. Other procedural imperative languages, such as C, FORTRAN and Pascal, can be supported with a more visible reliance on explicit system service calls.

Optimistic Coherence Model - The optimistic coherence scheme used applies only to data blocks actually accessed, and defers the required checking for write-set coherence until after the action completion. This separation of the execution of an action from the supporting task of maintaining its environment produces considerable tolerance of system latency. Additionally, this choice converts the problem of deadlock to a more tolerable form, livelock.

Uninterpreted Triggers - Many concurrent programming models (e.g., CSP [24], UNITY [14], and Orca [7]) require evaluation of a *guard* expression as part of the scheduling decision. The Affinity triggering mechanism forces the action code itself to perform any required conditional tests. This approach is flexible for the programmer and simple to implement. A less pedestrian consideration is that an implementation on a machine with heterogeneous processor types could benefit from the ability to make scheduling decisions without the necessity to interpret data structures used in the computational tasks.

Anonymous Triggering - An Affinity action with multiple triggers set cannot directly determine which one (or more) of the triggers has caused it to be scheduled. While it would not be difficult to provide such information, this would seem to detract from the nearly-stateless relational model of actions, and would compromise the system's freedom to replicate actions.

Non-preemptive Scheduling - Non-preemptive action scheduling allows a simpler kernel implementation and improved action efficiency, because it limits the variety of possible access conflicts in the node-data-block cache.

Number of Actions per Context - The binding of an action to a specific context is impermanent. The number of actions sharing a single context and the detailed management policies for contexts may affect the speed of execution of a correctly-written action (Section 4.4.1), but not its semantics. The current Affinity implementation can have multiple actions per context, but is generally used in a one-action-per-context mode.

4.1.1 Fault Tolerance

Fault tolerance, specifically the capability for transparent continuation of computations despite node failure, is a goal of the programming model. This implementation experimentally demonstrates the suitability of the model and the feasibility of supporting fault tolerance (Section 5.7), but no effort has been made to implement fault tolerance in a comprehensive manner.

4.2 S/2010 Hardware Characteristics

Some hardware details of the Ametek S/2010 are pertinent to the implementation discussion.

4.2.1 Message Network

The S/2010 message-passing communications network is based on a two-dimensional mesh interconnect of asynchronous routers [36, 39]. The *wormhole* routing protocol used preserves point-to-point message order, blocks for low-level flow control, and is reliable. The network routes messages from source to destination without interaction from intervening nodes, and reduces the costs of non-local communication.

4.2.2 Node Message Interface

Each computational node contains a microcoded communications processor to send and receive data packets to and from the message network without node processor involvement. The message-interface processor imposes a 256B maximum size on the body, or "payload," of data packets in the network. The processor sends and receives lists of packets and prepends and removes an additional 24B header containing routing and up to 14B of miscellaneous information. Short messages consisting of a header with a zero-length packet payload are used extensively in the Affinity implementation. The message-interface processor operates on buffer lists provided to it by the computational processor, which must perform periodic data buffer management. The message-interface processor has interrupt and scatter/gather capabilities, that link 256B packet buffers into arbitrarily long messages.

The nominal data rate is 25 MB/second each for send and receive, for an aggregate 50 MB/second communications bandwidth for each node. Higher-level software overhead reduces the sustainable rate substantially (see Section 5.3). Communications hardware latency is negligible compared to software latency.

4.2.3 Computational Node

The S/2010 computational processor is a 25 MHz Motorola 68020 microprocessor and Motorola 68881/2 floating-point coprocessor, with 4-8 MB of usually-zero-wait-state dRAM.

The custom virtual-address-translation unit has two page sizes: 8KB and 256B, the latter produced by an optional second-level translation that induces one wait state for processor access. The smaller page size is the same as the network packet size, and is used to allow physical scattered buffers to be made to appear contiguous in the action's

virtual-address space. The SRAM-based address-translation unit contains 4-1024 loaded contexts of 32MB-128KB virtual size.

4.3 Data Block Services

Data blocks are the storage entities underlying the object abstraction. When mapped into the virtual address space of an action, data blocks appear as compact linear segments of user-defined size. The minimum data-block allocation size is one small page, 256B. The actual physical storage is a list of generally discontiguous small-page buffers, which are processed by the S/2010 message system as needed for distribution throughout the multicomputer.

4.3.1 Data Block Master Copies

The Affinity implementation maintains a single master copy of each extant data block, which must be updated when actions act to change its state. This master copy is not generally in the same node as an action that creates or accesses it. The master copy is not directly accessible to an action; all reads and writes are performed on a local copy that is cached on the node in response to an action's attempted access. Even when an action uses a data block whose master copy resides on the same node, a distinct local copy is cached and used for direct access.

When a data block is cached in a node, a reference to that node is associated with the master copy. Modifications to the master data block are propagated to all nodes with recorded references.

A schematic view of a small computation is seen in Figure 4.1, showing two nodes of a multicomputer. Action A has accessed data blocks "P" and "Q," which has caused copies of each to be cached in the top node. The master copy of "P" is located in the bottom node, that of "Q" in the top. The dotted lines represent the logical connections between the cached and master copies of data blocks. It makes little difference whether or not the master copy is in the same node as the cached copy. Even intra-node updates are sent through the message system, rather than being treated as a special case, since the low-level message system is faster than a processor copying operation. The kernel components that maintain the cached and master copies are almost completely decoupled, as discussed in Section 4.3.3.

4.3.2 Node Data Block Cache

Each node maintains a large cache of data block copies that have been previously demanded by actions active on that node. This cache is a kernel software construct only, and has no special hardware support. The node-block cache is shown schematically in Figure 4.2. Two distinct action virtual address spaces are shown. Action A has references to blocks "P" and "Q," action B has references to blocks "Q" and "R."

Actions A and B both refer to a single physical copy of "Q" in the node-block cache. The decision to maintain a node-global data-block cache results in many simplifications to the cache maintenance code, and obvious savings in memory use when several node actions share common data block references. A potentially negative consequence of using a single copy of the block is that restrictions must be placed on concurrent access by

MASTER AND CACHED DATA BLOCKS

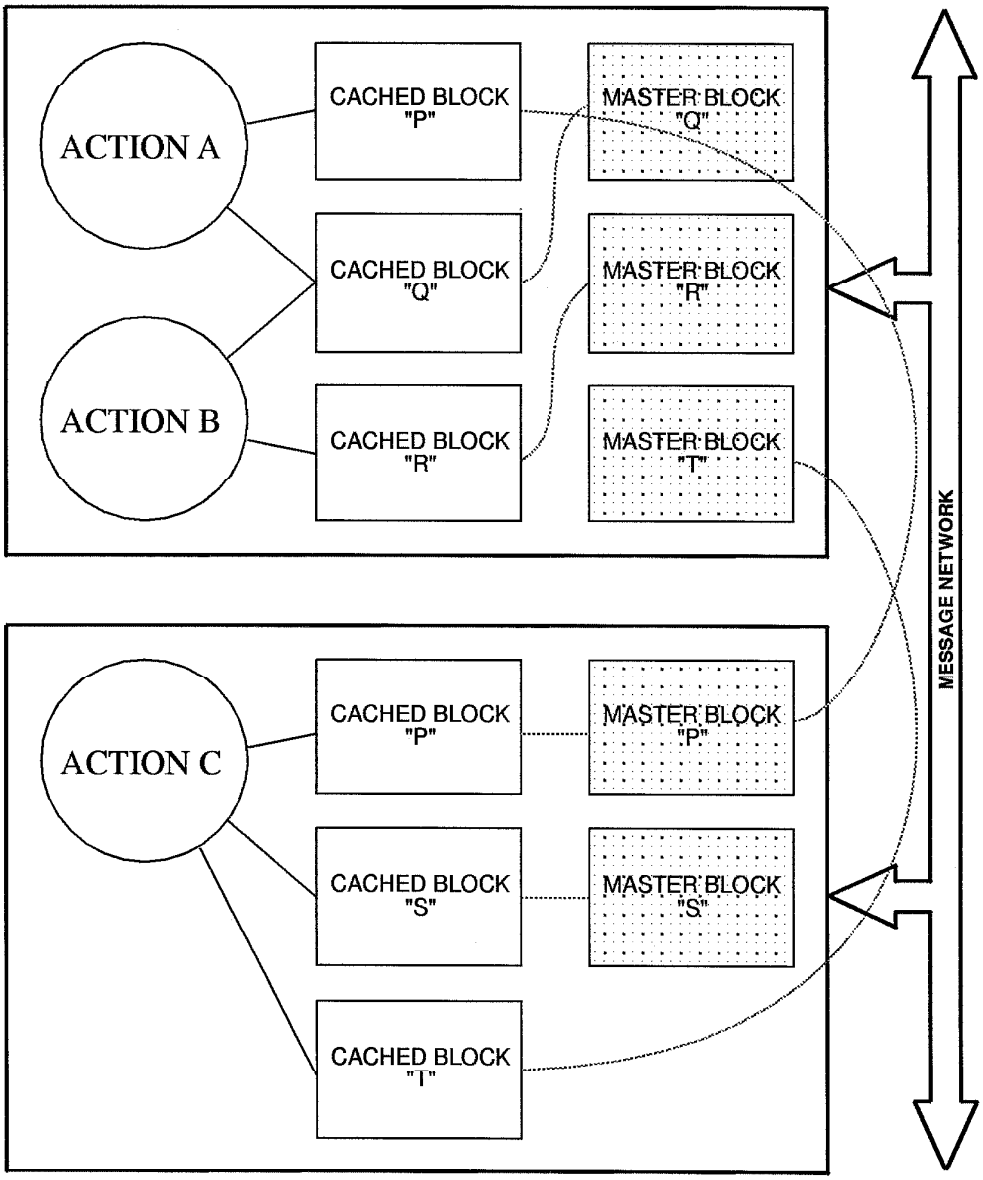

Figure 4.1: Master and cached data blocks on two multicomputer nodes. Action references cause data blocks to be read into per-node caches. Master copy location is arbitrary.

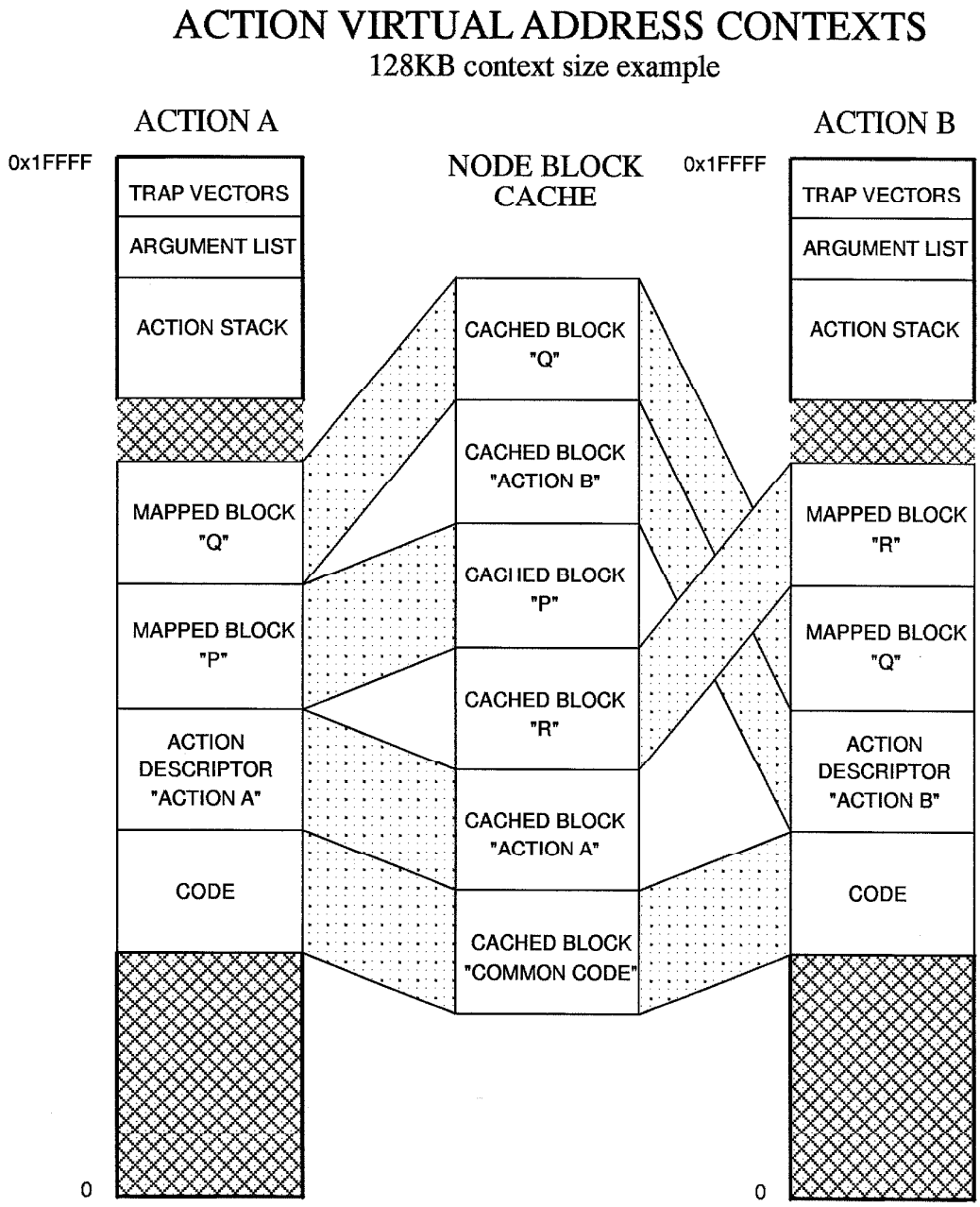

Figure 4.2: Two distinct action contexts, one action per context. Multiple contexts can map to shared data block copies in the per-node block cache. Only one action per node is active at any given time, limiting possible block cache access conflicts.

multiple actions. Allowing actions to run to completion without preemption simplifies the problem.

As implemented on the S/2010, the node data-block cache has common resource pools for both physical storage (the small page buffer pool used by the messaging system) and virtual address resources (the second-level address-translation mapping tables stored in the S/2010 address-translation hardware). The sharing of virtual resources is of interest because this permits two cache-maintenance functions to be managed globally for the node. First, when the content of the cached data block is updated by a new list of received message buffers, the task of remapping the existing segment of virtual addresses need be done only once. Secondly, when access to a data block is altered during the action execution cycle, the hardware page-protection fields need be changed only once. If the virtual address spaces of each action shared no resources, the job of maintaining current cache contents would be much more costly.

4.3.3 Asynchronous and Synchronous Services

The required Affinity data block services fall naturally into two classes. Operations on master copies of data blocks may be requested by another node, and require rapid response to maintain system performance and to minimize data staleness. These operations have no direct relation to the execution of the action (if any) currently active on the node containing the master copy, and are therefore referred to as *asynchronous* with respect to the action activation cycle.

Operations on a node's data-block cache are *synchronous* with action activation on that node. While an action is active (executing code), the node data-block cache is insensitive to changes in the master copy. The visible contents of each data block are effectively locked against "outside interference" while the action is executing. Of course, the action itself may modify the cached copy.

The natural division of asynchronous services operating on master data blocks and synchronous services on cached copies partitions most of the kernel code accordingly. There is no need for master data blocks to reside on nodes executing code; action and master-data-block location strategies can enforce complete distinction. Since the services required on the master data blocks are relatively simple, list-oriented operations, the nodes containing masters could be completely different from nodes executing user code in actions. A processor with fast interrupt service, no memory management, floating-point units, and minimal integer-arithmetic capability would suffice for the communication-oriented tasks of master services.

Affinity is typically configured to provide both asynchronous and synchronous services on each node. Since the S/2010 hardware provides a single queue for receiving incoming messages (see Figure 4.3), the low-level communications resources are shared by both sets of services. All asynchronous service-request messages cause an interrupt to the processor, which then digests all packets (partial messages) from the message receive queue. The packets are accumulated into complete messages, classified as asynchronous or synchronous by their embedded type field, and appended to the appropriate input queue. Asynchronous service requests are processed immediately upon receipt; synchronous services are processed between action activations. Almost all synchronous messages are requests to update the node block cache to reflect new versions of a master data block; the newly received message buffers replace the old buffers in the cache as

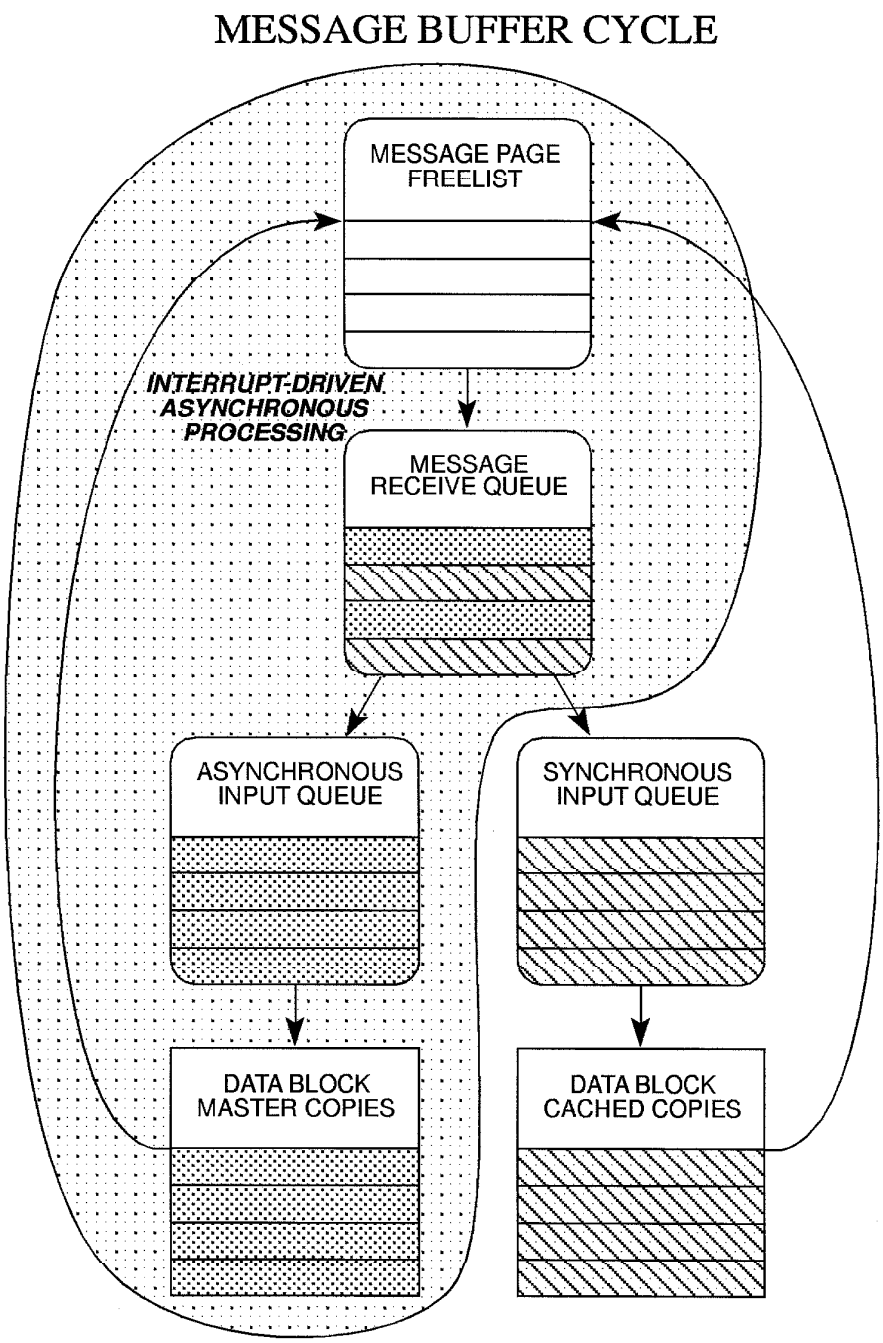

Figure 4.3: Asynchronous and synchronous data block services. Messages operating on data block master copies are processed asynchronously with respect to local action execution. Messages updating the node data cache are requeued asynchronously, but are transferred to the cache between action activations.

shown in Figure 4.3.

Some asynchronous functions do relate to actions. Although they could be implemented synchronously, there are performance advantages in processing them immediately. For example, a request to load an action on the node may immediately initiate a cache-loading request for the action-descriptor data block. This "prefetch" request will typically eliminate the effect of latency in the master-data-block server. Similarly, modified cached data blocks are locked on action exit, and unlocked by asynchronous postprocessing (see Section 4.7.2).

4.4 Action Context

Affinity actions are *anonymous*; there is no mechanism for actions to interact directly after the initial instantiation of one action by another. Specifically, unlike the Reactive Kernel, messages are not sent to processes. Actions are *asynchronous*; unlike systems based on remote-procedure-call (RPC) [10] mechanisms, there is no direct transfer of a thread of control between actions. Finally, actions are *atomic*; there is no visible effect on the computational state until a successful exit. These three characteristics, along with the semantics of instantiation and triggering, allow most functions required of actions to be implemented by standard data-block services. Few extensions to the user-level functions are required for kernel operations beyond allowing direct probes of data-block state and manipulation of MMU protection.

4.4.1 Action Transient Context

In the usual Affinity configuration, each action has a distinct context. Figure 4.2 shows the complete context required for action execution on a node. Most of the sketch shows the transient context, as defined in Section 2.10.1 the essential context is associated with the action descriptor (Section 4.4.2).

Transient context is an environment constructed to support action-code execution, using information derived from the action descriptor. The Affinity transient context consists of:

Block Mappings of the virtual address space to data blocks, including code,

A Stack Mapping to a common stack,

Argument List containing the values bound at action instantiation,

Trap Vectors that provide a mechanism for action exit.

In Figure 4.2 two actions are shown with their mappings to the physical buffers of the node data-block cache. Note that a block mapped in the distinct address spaces of more than one action context will, in general, appear at different addresses. Therefore, any pointer within a data block must be a portable-pointer type to have meaning in multiple action contexts. This also has significant implications for a kernel optimization, as discussed in Section 4.11 and below. The degree to which block mappings are retained between activations is an efficiency issue (see Section 4.10).

Stack

Since actions run to completion, without preemption by another action, a common stack can be used for all actions. This sharing significantly reduces the storage requirements and allows large numbers (hundreds) of actions to reside on a node. The current implementation pushes the argument list on the stack upon each action activation; to do otherwise would have required either multiple stacks or placing the argument list in a data block that would have used substantially more virtual and physical resources. Also, in the S/2010 processor, the two-level MMU introduces a slight (~5%) performance penalty for accessing data in small pages (256B) as compared to big pages (8KB). Since blocks are mapped as small pages, there is some advantage in not making the stack a data block, since it is a purely local and transient entity.

Code

Unlike the stack, action code is stored as data blocks. Standard data block operations provide for intra-node code sharing and distribution of code throughout the multicomputer. In the S/2010, a processor-hardware code cache minimizes the potential performance penalty associated with data-block access.

Action Descriptor

The action descriptor is a data block, created upon action instantiation, and contains a reference to the action-code block and the argument list bound at instantiation. Since a large argument list could cause the descriptor to exceed the 256B size of the physical message buffer, it is convenient to make it virtually contiguous by mapping. Making it a data block resolves this problem neatly.

4.4.2 Action Essential Context

As noted in Section 2.10.1, Affinity actions have very limited *essential* (semantically significant) context. Essential action context comprises the actual argument list, one bit of state indicating whether the action has run at least once, and references to the code and (optional) trigger-set data blocks, all located in the action descriptor.

4.4.3 The Trigger-Set

Actions that persist beyond one activation will have set triggers on at least one data block (Section 2.3). Since "one-shot" actions, e.g., root and spawned cut-out actions, (Section 6.3.1), do not set triggers, Affinity defers creation of a data block to record set triggers until required by the action. This record, the *trigger-set*, is part of the action's essential context.

4.5 Action Cloning

The consonance of kernel- and user-created components of action contexts allows the kernel great freedom in relocating and replicating actions. Changes to any data blocks in the action context, whether modified explicitly by the user or implicitly by the kernel,

will be applied atomically. Since the same mutual-exclusion and coherence guarantees hold for the action descriptor, trigger-set, and user-created data blocks, the semantic effect of an action is unchanged by either relocation to another node or replication of the action on other nodes. For example, if clonal copies of an action are instantiated on several nodes, only one will succeed in changing the "initial-execution" state bit held in the action descriptor. Because of action atomicity, the replication of actions can be viewed as essentially equivalent to the case of a single action rapidly relocated onto multiple nodes.

4.6 Number of Actions per Context

Many, if not most, concurrent-programming systems are implemented with all processes on a node sharing a single context. This allows implementors to layer their system atop an existing operating system such as UNIX or RK [45]. Since Affinity is a low-level kernel, the decision whether to have a distinct virtual address-space for each action can be made more freely than for experiments constrained by such layered implementations. The spectrum of design choices is not limited to a dichotomy between unit-context "lightweight" processes (e.g., Mach *threads* [55]) and UNIX "heavyweight" processes with completely distinct address spaces. The standard configuration of Affinity manipulates the processor MMU to maintain distinct processor address space contexts for each action, but neither processor state nor stack contents are saved between activations.

Distinct contexts offer protection against incorrect data access by unreliable code. Additionally, multiple smaller contexts containing only a few objects can allow simpler algorithms for management of virtual-address space than are appropriate for a larger, monolithic address space. The default action address space for the Affinity S/2010 implementation is only 128KB, 16 big pages, which allows a linear search for some virtual-address maintenance functions.

On the other hand, some relaxation of strict distinction between contexts can improve both space and time efficiency. It is highly desirable to maintain only a single physical copy of a particular cached object in each node, since this reduces both the physical storage requirements and kernel complexity. A further advantage is realized by sharing the object's virtual-memory mapping structures among all the contexts in a node, since the arrival of new versions of the cached object requires remapping to new physical buffers. This sharing of the cached virtual image of the object requires a MMU with multilevel mapping structures capable of sharing mapping information, such as the S/2010 two-level custom MMU or the tree-structured MMU tables of recent microprocessors such as Motorola's 68030/68040 and Intel's 386/486/860 products. Sharing both the physical and virtual images of a cached object is simplified by the requirement that Affinity actions run to completion.

The design of the S/2010 MMU makes it advantageous to use a 32-small-page cluster as the quantum of virtual-memory allocation. Virtual-memory-management efficiency considerations led to the decision to slightly reduce the read-access protection of objects. Since more than one object may be mapped in a cluster, it may be possible for an improperly coded action to read the contents of a theoretically inaccessible object sharing a page cluster with a mapped object. Since write-protection is carefully maintained, this is not a significant problem for a nonsecure computing environment.

4.6.1 Action Context Weight

While multiple contexts increase the storage requirements of the kernel, the total demand is comparable to more conventional lightweight processes: several hundred minimal actions can run on an S/2010 node with 4MB of memory. Support of virtual-memory allocation and deallocation is the major task associated with context management. Actual context switching is a relatively cheap hardware operation. Since the base implementation associates more virtual memory context but less processor state with its actions than is characteristic of "lightweight" processes, we might designate them as "welterweight" processes.

Affinity can place arbitrary numbers of actions in a single context. Generally a one-action-per-context configuration is used, but the option of a single virtual-address space for all actions on a single node does open the door for some performance optimizations (see Section 4.11).

4.7 Action States

Figure 4.4 shows the major states of Affinity actions. Actions are instantiated by the user's code, which binds the actual and formal arguments. This information is stored in an action-descriptor data block created by the kernel, and the new action request is enqueued for later placement on some node. If the parent action succeeds, a "load action" message is sent to some node. Note that there is no synchronization of execution nor value return between parent and descendent actions. Eventually, some node will *load* the action by creating a local context descriptor and caching copies of the code and action descriptors. On initial load, the action "comes up running," allowing it to set triggers. Subsequently, it must be triggered by a data-block modification to be *scheduled* for execution. The node sequentially removes scheduled actions from the queue for execution; the action currently executing code is *active*. The kernel may also choose to relocate a scheduled action by sending a "load action" message to any other node and deleting the local (transient) context, since all the essential state is stored "externally" in the data blocks. An action that fails (due to conflicts preventing coherent write-set update) will be rescheduled. A successful action with triggers set will return to the quiescent loaded state, waiting for another triggering event.

A triggering event for an already scheduled action is ignored. There is no queueing of triggers for a particular action; transition to the active state satisfies all prior triggering events. A successful action without triggers will be deleted.

4.7.1 Action Abortion

It is convenient for both the programmer and the kernel to be able to force an action to fail. Both do so by adding into the action's write-set a special-purpose data block (Section 4.12) that is permanently locked and cannot be successfully modified. For efficiency, the action's execution is stopped immediately, but no other deviation from normal action postprocessing is required.

The `s_abort_action()` system service introduced in Section 3.6.3 can be invoked when the user code determines that no useful work can be performed; the action is

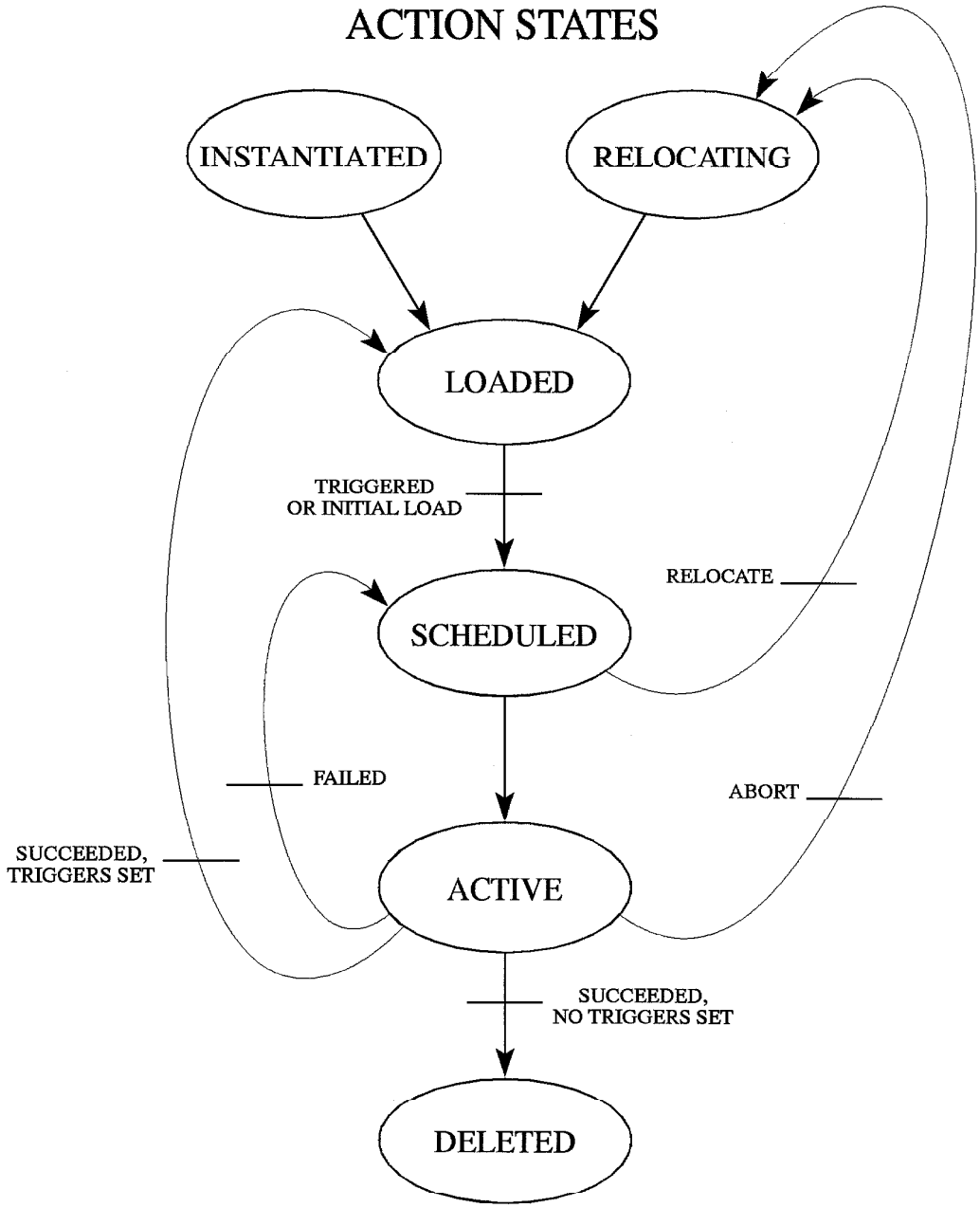

Figure 4.4: Action States

rescheduled with no effect on computational state. An abort, like any other failure, causes immediate rescheduling.

The kernel uses the same mechanism to implement growth in virtual context size. The S/2010 address-translation hardware has several fixed context sizes. At instantiation, there is no way to usefully predict the ultimate size of the action's virtual-address space, since it grows as blocks are referenced. By default, a minimal context (128KB) is allocated. If the action grows to require a larger context, it is quite easy to provide it simply by aborting the action and sending a load-action message with a larger suggested context to a node.

Since the local transient context is dispensible, this technique can be used freely, for example, to help purge the node data cache (Section 4.10).

4.7.2 Action Blocking

Figure 4.5 shows a somewhat more detailed view of the execution of the activated action, also showing the interaction with the node data-block cache processing discussed in Section 4.8. This figure shows two circumstances that may cause the kernel to block waiting for a response from another node. The code and action descriptors will be fetched if they are not present in the node data-block cache, just as in a typical virtual-memory demand-paging operation. Also, cached blocks may be locked pending resolution of a prior action's success or failure. Neither cause of blocking causes significant waiting in normal use (see Section 5.5).

Since the block update protocol is substantially decoupled from action activation, the two activities can be performed concurrently. Action scheduling and activation is overlapped with postprocessing of the effects of prior actions. A scheduled action can be activated before the success or failure of the just-terminated action is decided. Only if subsequent actions access the locked blocks before the decision will there be a wait. The technique is very effective; in practice, computations rarely experience any blocking after startup.

4.8 Data-Block Modification

Modification of the contents of data blocks by actions is the basic mechanism for computation and communication in Affinity. These data-block operations are intimately connected with the action activation cycle discussed in Section 4.7.

4.8.1 Dirtied Write Sets

Actions act directly on the contents of local copies of data blocks, the "current versions" in the node data-block cache. When a data block is written to by an action, it is presumed that the content is altered, and the block is marked as *dirty* in the node block cache. All blocks written by an active action are added to the current write set. After the action terminates, the kernel attempts to write back to the master data blocks the new versions of the write set. A record of the write set produced by this particular action activation is retained until the kernel can decide whether these changes can be performed successfully. Until that decision is made, the cached data blocks in the dirty set are locked against access by any action. Attempted access to a locked block will

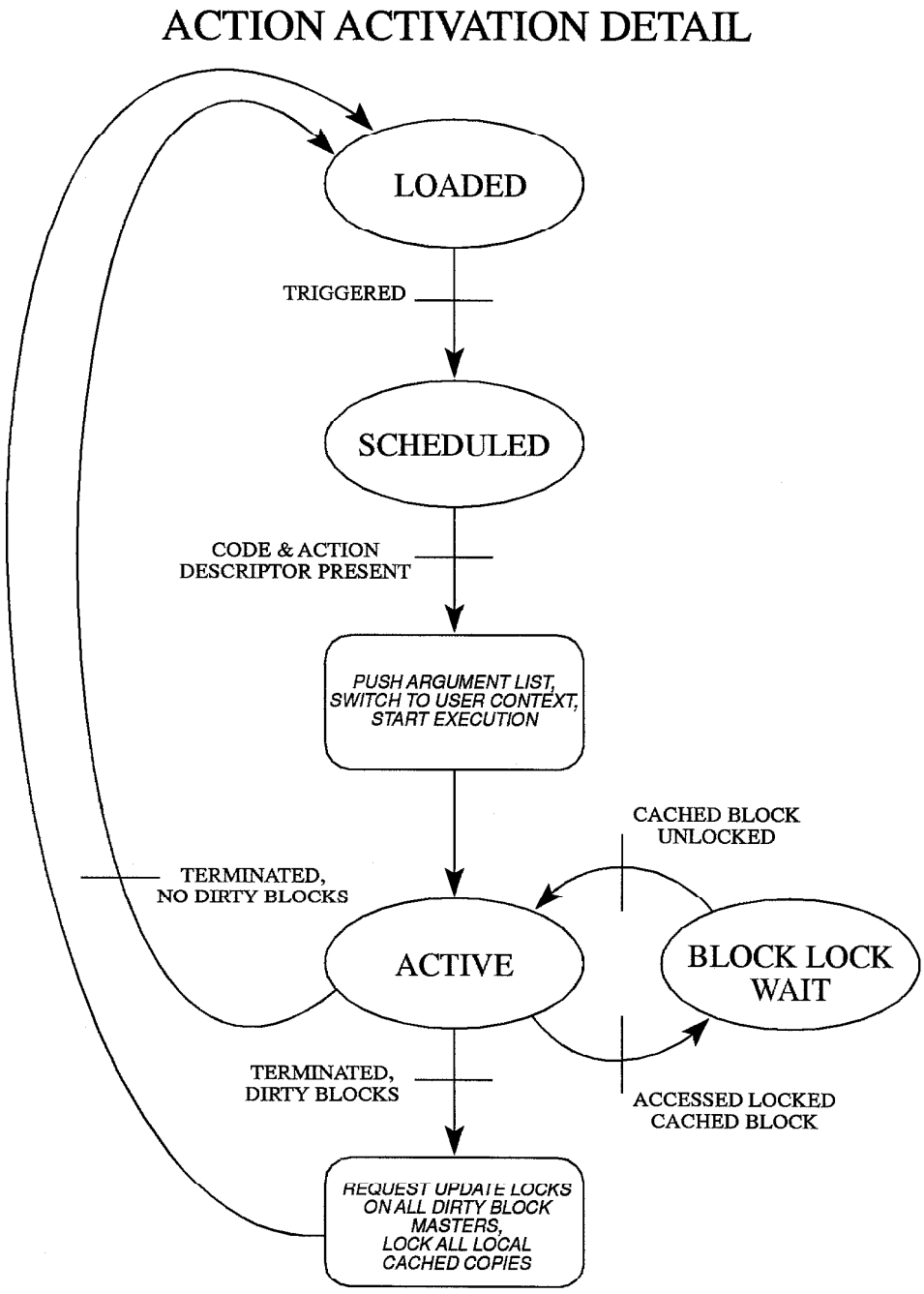

Figure 4.5: Action activation detail. A write to a data block makes it "dirty." Cached blocks may be locked pending resolution of a prior action or for a demand fetch.

cause the action to block, waiting for the decision and subsequent unlocking of the cached block.

4.8.2 Data-Block Versioning

Both the master and cached copies of data blocks contain monotonically increasing version numbers. Each time the data block is dirtied by an action, the version number of the cached copy is incremented. Affinity semantics require that an action modify the current version of a data block. Operationally, this means that an attempt to write back a dirty cached block to a master copy can succeed only if the "clean" cached block version was identical to the master, and the dirty version is therefore one more than the current version of the master. If multiple actions concurrently write a block, at most one will succeed.

4.8.3 Data-Block Update Protocol

A simplified diagram of the data-block update protocol is shown in Figure 4.6. The left half of the figure shows how the kernel works to maintain coherence between dirtied blocks in its local node data-block cache and the master copies of those blocks. The right half shows how a request to update one particular master copy is processed. In general, an action activation will dirty multiple data blocks, and multiple update requests will be sent, so it should be understood that the figure represents only one of several master-copy update requests. The solid vertical arrows represent a simplified flow of control; the dotted horizontal arrows indicate a message transmission.

4.8.4 Action Postprocessing

After an active action has exited, the number of cached data blocks dirtied by a write is checked. Zero dirty blocks imply immediate action success, albeit with no effect. A nonempty write set initiates the two-phase [35] block-update and action-success decision process sketched. "Update-lock" requests are sent to nodes containing the master copies of dirtied cached blocks. Such requests will be granted if the action modified the current version of the block and the master is not currently locked by another node's prior request; otherwise it will be denied. This decision is trivial and the reply is immediate. If all requested master locks are granted, the action has succeeded; if not, it has failed, and will have no visible effect.

While this two-phase update protocol does involve locking the master copies, it is important that it does not do so while the action is actually executing, potentially a long time. The master copy is locked only for the time required for the cache processor to decide whether it can obtain write locks to all members of the particular write set; this processing is asynchronous and quick.

4.8.5 Optimism Invites Disappointment

This optimistic scheme achieves maximal concurrency and reduced sensitivity to latency at the risk of discarding the results of actions with write-set conflicts. For computations with substantial inherent concurrency, this is a good choice; algorithms with poor concurrency will tend to find their "natural level" of concurrency by experiencing high

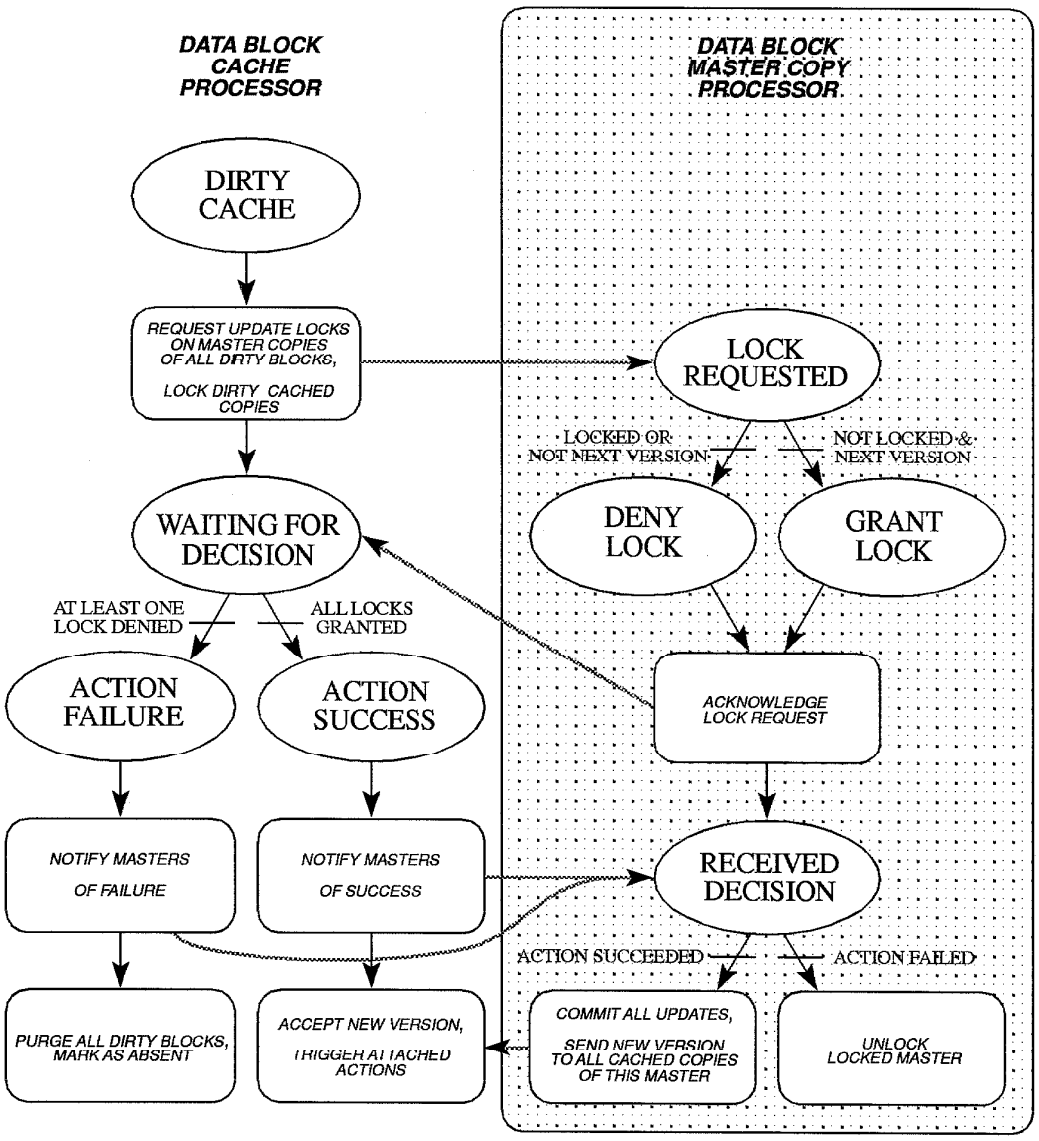

Figure 4.6: Block update protocol

action-failure rates. The alternative approach, locking the block for the duration of action activation, would stall other actions accessing it, and cause serialization at best, deadlock at worst.

Messages indicating success or failure are sent to the nodes containing locked masters. Successful actions cause acceptance of the requested update to the master copy. The new versions of the data block are then sent to all nodes that have cached a copy.

The cache processor portion of Figure 4.6 is somewhat oversimplified. The initial action postprocessing, requesting update locks on the master copies, and locking the dirty cached copies, occurs synchronously, interleaved between action activations. The decision of action success or failure is asynchronous, and will not affect activation and execution of a subsequent action unless that action accesses a cached block that is still locked pending a decision on success or failure of the prior action. Such an access will force a *block lock wait* until the decision is made, as was shown in Figure 4.5.

The last operations of the cache-processor diagram require some elaboration. On the path of action success, accepting a new version of the data block (consequent to a master-copy update) unlocks the dirty cached copy. On the path of action failure, the dirty cached set will be unlocked, but marked as *absent*, and the dirtied contents will be discarded. Absent blocks may be revalidated by acceptance of a new version of the data block. Alternatively, if an action attempts to access an absent cached block it will be forced to wait while a copy of the current version is demanded of the corresponding master.

In general, acceptance of a new version of a data block into the node data-block cache is interleaved between action activations. A minor special case is when the local cached copy was dirtied by an ultimately successful action. In this case the "acceptance" occurs asynchronously when the "decision" lock is removed.

4.8.6 Data Staleness

When actions become active, the data-block versions current in the node data-block cache at the instant of activation are the ones upon which all calculations and modifications are made. If other actions succeed in writing the corresponding master data blocks during the action's execution, the action calculations are based on out-of-date, *stale* data. If a member of the write set is stale, the action will fail. Staleness of the read-set may or may not be significant, depending on the details of the computation.

4.9 Deadlock, Progress, Fairness

The particular scheme chosen for data-block coherence is optimistic in that it assumes that an action will be able to successfully write all data blocks desired. This will not necessarily be the case if the block has multiple concurrent writers. For computations with limited write conflicts, this minimizes latency due to data distribution or remote-access requests. Equally important, it eliminates the possibility of deadlock. An action that cannot write all desired data blocks will fail, to be immediately rescheduled for execution. The kernel need not deal with the problem of deadlock; however there is presently no way to guarantee progress of a computation with inherent write conflicts. In practice, even computations with severe conflicts do make slow and serialized progress,

though node success rates can become very unfair.

There is no guarantee of fairness in action scheduling. This implementation uses FIFO scheduling on each node, but the possibility of repeated action failure due to write-set conflicts makes it impossible to guarantee action success. For computations that terminate, the reactive programming style tends to minimize the significance of this issue.

4.10 Purging, Paging and Deletion

The node data-block cache contains three kinds of information.

Buffers - from the message system containing the actual contents of the cached data blocks.

Mappings - from mapped sets of message pages to the physical buffer addresses.

Control Information - recording the reference to the master copy of the data block, all contexts that have mappings to a given block, actions with set triggers, and current access and update states.

The node data-block cache can become quite large (1MB in the present implementation) and may contain data blocks not recently used. Much of the physical storage can be reclaimed if needed. All cached copies except for the few that are locked by a pending update cycle can be *purged* by freeing the buffers, invalidating the virtual to physical mappings, and marking them as absent. The mapping resources and control information are retained. If an action requires access to an absent block, it will fault upon access and a copy of the current content will be fetched from the master copy, as with standard virtual-memory "demand paging." Since this activity will result in a delay while the node containing the master serves the request, aggressive cache purging will generally slow computation and is to be avoided. The circumstances under which eager purging of the node data-block cache could improve performance (frequent modification of little-used blocks without triggers) appear to be rare; lazy purging is standard.

A reference count is kept of the number of action contexts that map each cached data block. When the count drops to zero, the block may be deleted entirely from the node data-block cache, reclaiming all resources and removing the record of reference in the master copy. This eager deletion has been found to be generally deleterious to performance, since it is common for an action, particularly a root action, to pass a reference to a newly created data block to a newly instantiated action. If the root action does not set any triggers (the usual case), it will be deleted after termination. An eager deletion policy would allow the cached copy of the new block to be deleted, since no actions (specifically *loaded actions*, see Section 4.7) would have it mapped. However, as soon as a newly-spawned action accesses the block, it would be forced to wait as a new copy of the just-deleted block was fetched into the cache. A similar problem can exist for blocks containing action code, which are commonly shared, but could be subject to deletion during startup. Therefore, the standard policy employed is lazy purging and lazy deletion of the node data-block cache, with periodic or demand-triggered sweeps as needed to reclaim storage.

4.11 Reference Localization

Affinity portable pointers are implemented as a C++ class containing the location of the master copy of a data block. When the portable pointer is dereferenced to access the data block, it must be converted to a standard C++ pointer to a virtual address in the action's context. The dereferencing may require a system-service call to either map the data block if it is not already mapped into the context, or to find the block's address if it is already mapped. The system service is reasonably efficient, using hashing to determine whether the block is in the node cache, and linear search of the (usually small) context to find the mapped address. Nevertheless, this dereferencing trap will be much slower than use of a standard pointer, and can significantly slow action execution. In fact, the portable pointer has a "localized" form, which contains the virtual address at which the block is mapped. The address of the localized form is context-specific, which limits its utility. In general, the kernel cannot localize references held in data blocks, since these may be shared with actions in other contexts.

4.11.1 Localized Argument List

In two related cases, references can be safely localized by the kernel. The most important case is portable pointers in the action argument list. Since this localization need only be done once for a given context, code that dereferences a portable-pointer argument will run at speed after the first use in the initial activation (unless the action is relocated, etc.). Likewise, portable pointers stored in local variables that are automatically created on the stack when actions are entered (*auto* variables [18]) can be localized without concern, since the stack content is meaningful only during a specific action activation.

4.11.2 Localization in Data Blocks

Other optimizations are possible, at increasing complexity and cost. If all actions on a node share a single address context, it is possible to localize references contained within data blocks. The problem is that such localized references must be converted back to a portable form if the data block is successfully modified, since the data block may be used in another node. This optimization could be managed either by maintaining a local history of reference localizations or by a more general tracking of reference propagation (see Section 6.4.9). It is unclear whether this optimization is worthwhile for most medium-grain programs, particularly since even nonlocalized references tend to be cached by either compiler temporaries or C++ reference aliases.

4.12 Special-Purpose Kernel Data Blocks

A few special-purpose data blocks are created by the kernel at boot time, with the master copies allocated locally on the node. For example, as previously mentioned in Section 4.7.1, action abortion is implemented by attempting to write to a permanently locked block. Each node has blocks for termination detection and local kernel information (e.g., clock-tick count, performance statistics). While these data blocks may be efficiently read by action code, they are not user-writable. The system-information

block bypasses part of the standard master-modification cycle to allow it to be updated efficiently.

4.12.1 Detection of Quiescence

The easy interface to kernel information allows some tasks often incorporated into a kernel to be performed by user-level actions. The `checkTermination` library class introduced in the example of Section 3.8.4 instantiates a user-level action to monitor node activity. It sets triggers on all of the per-node local termination blocks and examines the per-node kernel information blocks to evaluate quiescence. The result is summarized to an interface data block that can be used to trigger printing of results, etc. Performance statistics are similarly gathered by user-level actions.

4.13 Action Scheduling

Each node places actions on a local scheduling queue in response to a triggering event or load-action message from another node. A node receiving a load-action message may forward the message to another node, for example, to improve load balance. No record of the action's placement is retained in either the originating or forwarding nodes.

4.13.1 Priority Scheduling

Affinity does support non-preemptive, multiple-priority action-scheduling queues. Each node contains a kernel-spawned printer-server action that uses a version of the producer/consumer buffer class (Section 3.6) to ensure that an action's call to `printf()` will have effect only for successful actions: a message sent directly to the host console server could not be cancelled in the event of action failure. The printer server is a high-priority action, so that it may empty the local print buffer quickly. A low-priority queue has been used for experiments with redundant, replicated actions (see Section 5.7).

4.13.2 Non-Preemptive Scheduling

Affinity does not currently support preemptive scheduling of actions. (Interrupt-driven kernel functions, e.g., master data-block services, are not actions and are preemptive.) Each preempted action would require considerable saved state, including processor state and a stack image. This amount of state is much larger than is required for non-preemptive scheduling, and could dramatically reduce the number of actions that can be loaded on a node. A general policy of action preemption, e.g., time-slice scheduling, would be prohibitively costly for the desired granularity of Affinity actions. More limited preemption would be feasible. For example, the storage requirements would be modest for a policy of priority-based preemption, i.e., non-preemption within a given priority set, but preemption of lower-priority actions. The sticking point is that, lacking restrictions on data-block sharing across preemption sets, it would be necessary for the kernel to ensure that concurrent actions on a single node did not violate Affinity action-atomicity guarantees by modifying shared data blocks in the common node data-block cache. In general this would require faulting on an initial block read access, which would

penalize all actions. More specialized use of action preemption might be justifiable (see Section 6.4.3).

4.14 Cloning Policies

While Affinity semantics allow actions to be cloned, some restrictions are desirable. The **hello1** example of Section 3.2, Figure 3.1, is not well-defined since operations such as printing to the console are outside of the programming model. Should we clone each of the four "printer" actions? It would not necessarily be incorrect to print several copies of each greeting, but it probably would surprise and annoy. Note that this particular problem doesn't exist for **hello2** or **hello3** (Figures 3.3 and 3.7), since clones of those actions would share and decrement the `lines_left` counters. The implicit mutual exclusion would produce the expected printing sequence. Actions that always operate with state alteration within the atomic-assignment model can be cloned with impunity. Operations outside of the assignment model, such as printing and action spawning, are problematic, particularly if there is not a corresponding data-block modification.

A couple of observations can be made about when cloning is likely to be helpful. "One-shot" actions are obviously not good candidates for cloning. A no-cloning policy for this category covers the cases of root actions and the potential embarrassment of **hello1**. Likewise, actions that have requested a fixed node assignment by a variant `spawn` (see Section 3.8.3) for timing purposes shouldn't be cloned. These two implicit criteria produce reasonable results for all the example programs. It may be possible to conjure up programs with more dynamic behavior for which explicit control of cloning might be useful. However, since potentially troublesome activities such as printing or spawning can be logged to shared variables, such a function can be handled by user code, and is probably not worth arrogating to the kernel.

Chapter 5

Performance

Previous chapters have introduced the Affinity programming model and discussed the Affinity implementation for the Ametek S/2010 multicomputer. This chapter presents some statistical measures of performance to demonstrate that this implementation is reasonably efficient and scalable.

The Affinity kernel implementation is a research tool for which simplicity is more important than absolute efficiency. However, the efficiency is quite respectable and compares favorably to other reported results.

In addition to making the model and implementation credible, performance observations can provide insight into both program and system design. The All-Points Shortest Path (APSP) program is examined in considerable detail, exposing the character of its structure as well as that of the supporting hardware.

This chapter is necessarily specific to the S/2010 implementation and its hardware, but the results provide a basis for evaluating the suitability of other platforms for implementing of the Affinity model. The data, with some extrapolation, allow us to consider concretely the limits of the medium-grained approach to concurrency inherent in this implementation and, to a lesser degree, the model.

Finally, the use of action cloning as a mechanism for fault-tolerance is examined in Section 5.7.

5.1 Speedups

Figures 5.1 and 5.2 show program runtime as a function of the number of multicomputer nodes, N, for the Affinity matrix-multiply and APSP examples of Sections 3.8 and 3.9. Log-log graphs are used because an ideal linear *speedup* will be plotted as a straight line. Speedup, defined as the ratio of runtimes of single-node and multiple-node executions of a given program, is a popular metric for evaluating the performance of concurrent systems. While intuitively appealing, speedup is potentially misleading [56] because concurrent formulations may suffer significant additional overhead relative to good sequential implementations. The increased program complexity required to explicitly manage data distribution for a concurrent formulation of a program commonly slows single-node execution by a factor of two or four [53]. If measured only with respect to the single-node performance of the concurrent version, speedup will overstate the effective performance experienced by the user. To ensure that we are not confronting a

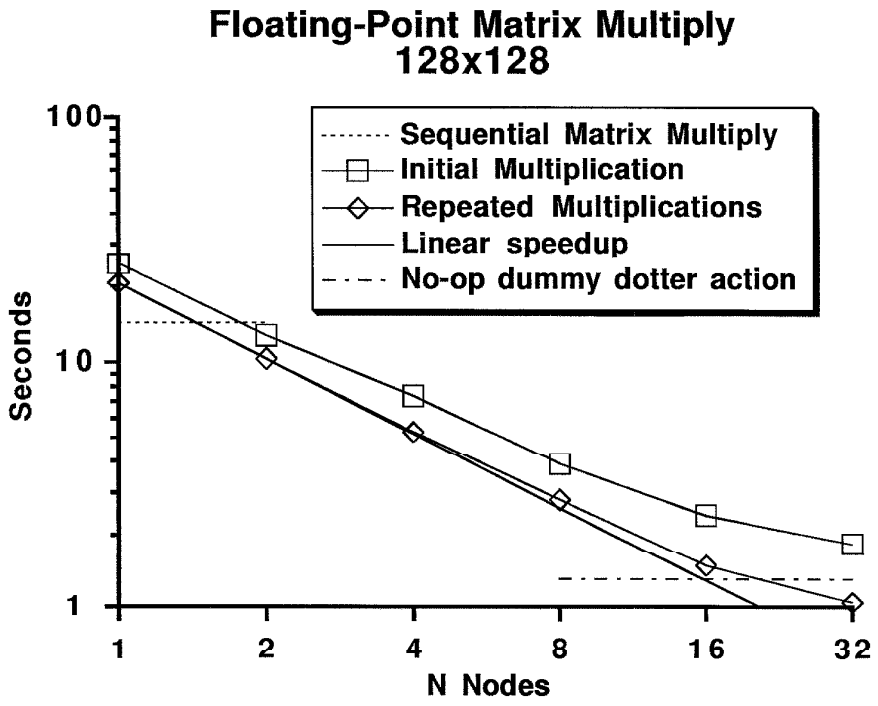

Figure 5.1: Speedup for the floating-point matrix-multiply program example of Section 3.8. The concurrent and sequential versions of the program run the same source code, making for a fair comparison. The two-node concurrent version runs faster than the one-node sequential version, showing that the concurrent environment is reasonably efficient.

straw man, both of the runtime-versus-N plots show the runtimes for good sequential versions of the program.

Though the concurrent versions do show lower single-node performance, Affinity is efficient enough that the slowest truly concurrent cases (running on two nodes) are faster than the sequential versions of the programs. A user porting code from a workstation will find a modest concurrent speedup infinitely more gratifying than an initial concurrent slowdown.

A practical objection to the speedup statistic is that it may be difficult to run the same program over a wide range in the number of nodes. A program small enough to fit on a single node may run so quickly on a large configuration that accurate measurement becomes difficult or ambiguous. We shall follow the speedup discussion with more detailed direct measures of system activity that provide a firmer foundation for analysis.

5.1.1 Experimental Configuration for Speedup

All experiments were run on a 32 node S/2010 multicomputer. For the speedup measurements, all nodes ran actions and also provided master-copy services. To improve timing

accuracy, one node was dedicated to the root, termination-detection, and result-printing actions, so only thirty-one nodes ran "productive" actions for the most concurrent case.

5.1.2 Matrix-Multiply Speedup

As mentioned in Section 3.8.4, the matrix multiply program can be run as either a sequential or concurrent program without change in the code, making the single-node comparisons quite accurate. The sequential version computes the dot products by making function calls instead of spawning "dotter" actions. The dashed line on the left of Figure 5.1 shows the runtime of the sequential program on a single node; it is extended to the right to show the intersection with the concurrent-version runtime lines.

The "Initial Multiplication" line shows the runtime for the first computation of the matrix product. To produce the "Repeated Multiplications" data, the program was altered to touch each row of the input matrix A (see Figure 3.23) after completion of each multiplication, triggering a recalculation. The longer time required for the initial multiplication represents the additional cost of spawning 128 "dotter" actions, performed sequentially by the root action. The lower dashed line shows the cost of spawning dummy "do-nothing" actions for a 32-node configuration. These dummy actions are not as fully instantiated as the real "dotter" actions because they do not set triggers and are marked for deletion after one execution, but this datum gives a lower bound for the runtime of the 32-node "Initial Multiplication." Since this point is above the corresponding "Repeated Multiplication" datum, in the most concurrent case, the initialization cost of the problem exceeds the time devoted to arithmetic.

In the one-node case, the longer time required for subsequent concurrent multiplications, compared to the sequential version, is due to the relatively greater cost of scheduling actions in place of procedure calls, as well as the data-block page fault and post-processing. The costs of distributing the data across the machine are less significant, but do account for part of the slight upward inflection away from ideal linearity at the right end of the "Repeated Multiplications" line. Other minor sequentialities, such as a non-concurrent implementation of termination detection, can also start to stand out as runtimes become small, and definitional issues ("What is the concurrent part?") can dominate the measurements. While the speedup graph is instructive to a point, it is difficult to generate significant data across a wide range of configurations.

The matrix-multiply example illustrates how a familiar problem may be readily cast into the Affinity mold, but is relatively uninteresting for detailed performance analysis because there is no communication between the initial distribution of actions and data and the delivery of the result matrix. We therefore focus on the APSP program.

5.1.3 All-Points Shortest Path Speedup

Figure 5.2 shows the all-points-shortest-path program's (Section 3.9) speedup, along with a good sequential-program reference point. The sequential program uses a heap-based implementation of Dijkstra's single-point-shortest-path (SPSP) algorithm [2] run for each vertex of the problem graph to generate the all-points-shortest-path (APSP) matrix. The sequential startup costs have been eliminated from the concurrent program runtimes, showing a nicely linear speedup of the concurrent computation. The speedup between the single- and two-node cases is anomalously divergent from the linear-speedup

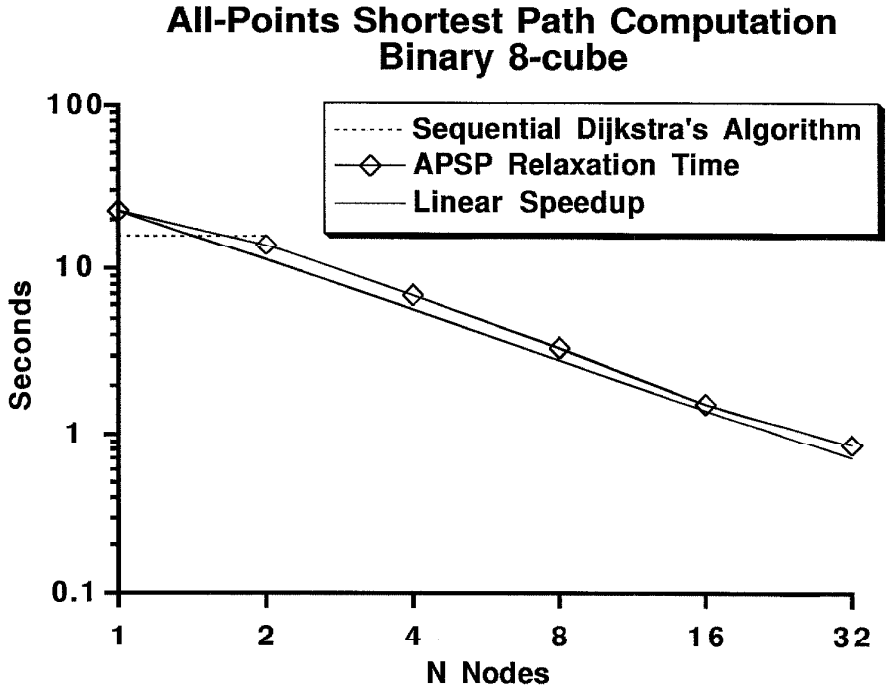

Figure 5.2: Speedup for the All-Points Shortest Path program of Section 3.9. Startup costs have been subtracted from the runtimes, so only the concurrent "relaxation" to the fixed-point shortest path result is measured.

slope. This aberration is apparently an artifact of a slightly advantageous sequential action schedule; the single-node case executes about 10% fewer actions than the two-node configuration.

The concurrent APSP program is comparably efficient to the sequential algorithm in this example, and is simpler. The choice of a logarithmic-diameter graph such as the binary cube is admittedly beneficial to the concurrent formulation; we can generally expect the number of actions run to increase in proportion to the graph diameter. The sequential algorithm would doubtless do much better for large-diameter graphs such as rings.

5.2 Node Processor Activity

Figure 5.3 is a time-windowed snapshot classifying the average node processor activity in the most productive phase of the APSP computation. The statistics are gathered by trace code in the kernel. The action time can include kernel operations directly invoked by the user code, specifically, page-fault and system services, such as new data-block creation, trigger setting, and reference localization. However, the mid-phase shown is free of all but page-fault overhead.

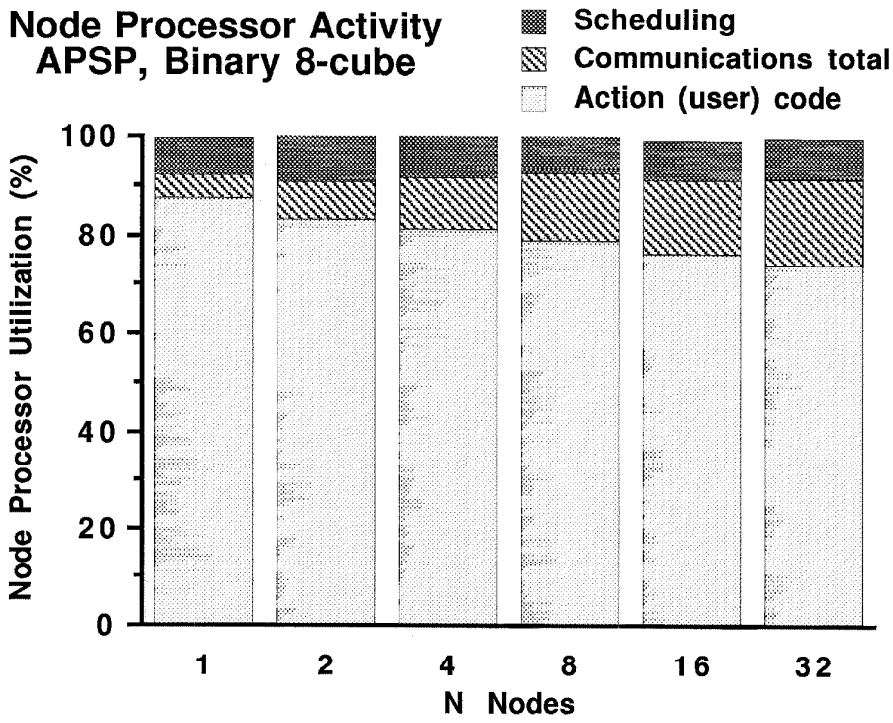

Figure 5.3: Snapshot of node-processor activity for the All-Points-Shortest-Path program during mid-computation. The action-scheduling cost is essentially constant, but the node processor's communication-related costs grow slowly with increasing concurrency as data is more widely distributed. There is no null time nor any blocking on inaccessible data, showing that system communications latency is completely hidden.

Each invocation of an APSP "vertex" action performs a nearly fixed number of operations. Since the rate of action execution is nearly constant, the scheduling costs, i.e., selecting a ready action, pushing arguments onto the stack, switching context, and later deciding action success or failure, are also uniform over variation in the number of nodes. The communications cost has several components summed together to represent the processor cost of all messages sent or received. The fraction of time devoted to executing user code is a measure of the program's efficiency, ranging from .875 to .743 for these samples. The processor has no idle time, nor is it forced to block while waiting for messages from other nodes, indicating that system latency has been completely hidden (see Section 5.5).

5.3 Communication Rates

The S/2010, as the first of the second-generation multicomputers, combines a communications subsystem dramatically faster than prior machines with a processor of comparable performance. The resulting machine is imperfectly balanced, and its over-

all communications performance is limited primarily by the CPU's inability to perform low-level message processing at the rate required to sustain the 25 MB/second hardware rate. Lack of raw processing power and a network interface design requiring some per-packet software maintenance effectively limit the sustainable low-level communications rate to about 12 MB/second. About 5 MB/second is the maximum seen in an Affinity "ring test" program sending large data blocks without processing. However, we shall see that the performance of programs performing any significant processing of their data is dominated by computational rather than communication costs.

5.3.1 Experimental Configuration

For all but the speedup measurements, the S/2010 nodes were segregated by function. Four nodes performed data block master-copy services only, one was dedicated to the root, termination-detection, and performance-monitoring actions, and the rest ran the other actions without any master-copy duties. This configuration makes for convenient analysis of the distinct communication sources. The reader should be aware that the nominally 32-node case therefore has only twenty-seven nodes running "productive" actions; no statistical adjustments are made for this distortion.

Each node maintains a data block with current kernel statistics, which may be accessed by any action. Although the local copy is updated continuously, copies are released to the rest of the system at a low rate, 5 Hz for these experiments. To produce time-series data, some high-priority performance-monitoring actions copy the contents of the system information blocks into FIFO buffers (one per node) while lower-priority actions compute the delta values from the input stream and aggregate them into time-windowed bins. The system is simple, but since we restrict these actions to a single node to reduce the system impact of our measurements, it is possible to saturate the performance-monitor node. This scheme can monitor all the system activity up to the sixteen-node configuration, i.e., sixteen action-only nodes, four master-service-only nodes, and the monitoring node's own activity. The most concurrent configuration can saturate the monitoring node, causing loss of timing information though the aggregate data remain accurate. The N = 32 configuration therefore typically monitors only sixteen of the action-only nodes; this limitation does not appear to introduce any significant error.

The following data are selected from the time-series as representative of the most productive phase of the APSP computation. Since the initial execution of the actions performs some house-keeping chores, e.g., setting triggers, the time sample selected follows the action's initial activation, generally about one-quarter of the way through the computation, later for the two most concurrent configurations. The time-series windows were 500 milliseconds for the long-running one- and two-node cases, and 200 milliseconds otherwise.

5.3.2 Aggregate Network Utilization

Figure 5.4 shows the total traffic for the S/2010 network during the APSP 8-cube computation. The utilization is a small fraction of the message-routing hardware's capacity, but, in the most concurrent case, the four master-service nodes saturate for much of the computation at per-node rates somewhat above 1 MB/second. The 8-

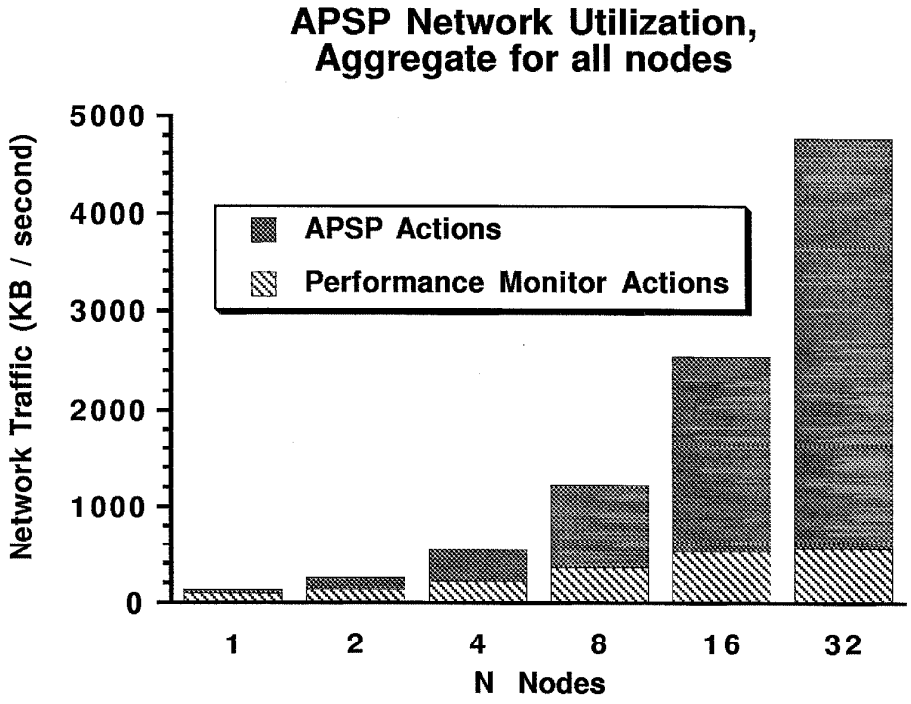

Figure 5.4: Aggregate network communications rates for All-Points Shortest Path computation for binary 8-cube graph.

cube problem generates minimum-sized data blocks of 256B; the master-node saturation bandwidth would increase significantly for larger data blocks.

The average size of the message payload increases with the number of nodes, from 131B up to 200B. The implementation's two-phase master-copy update protocol sends one header-only message and one copy of the data block from the node running the action to the master-service node, so the typical lower bound for the average size of the message payload will be one-half the size of the data block. In this example, the lower bound is 128B, quite close to the observed value of 131B. The average size grows as the master-service nodes are required to redistribute the new version of the data block to more nodes, a "multicast" operation (see Section 5.3.3).

For this mix of master-data-block services, at saturation, each master-service node is processing a complete master-copy request/lock/grant/update/multicast cycle every 1.8 milliseconds, a data-block send or receive every 240 microseconds, and a message (either a data block or a short, header-only message) every 166 microseconds. These numbers compare favorably to other low-level multicomputer DSM implementations [51], and are many times faster than DSM implementations and distributed-object systems on workstation platforms [3, 25].

The performance monitoring actions generate considerable network traffic, but are limited in the largest configuration by saturation of the one dedicated monitoring node

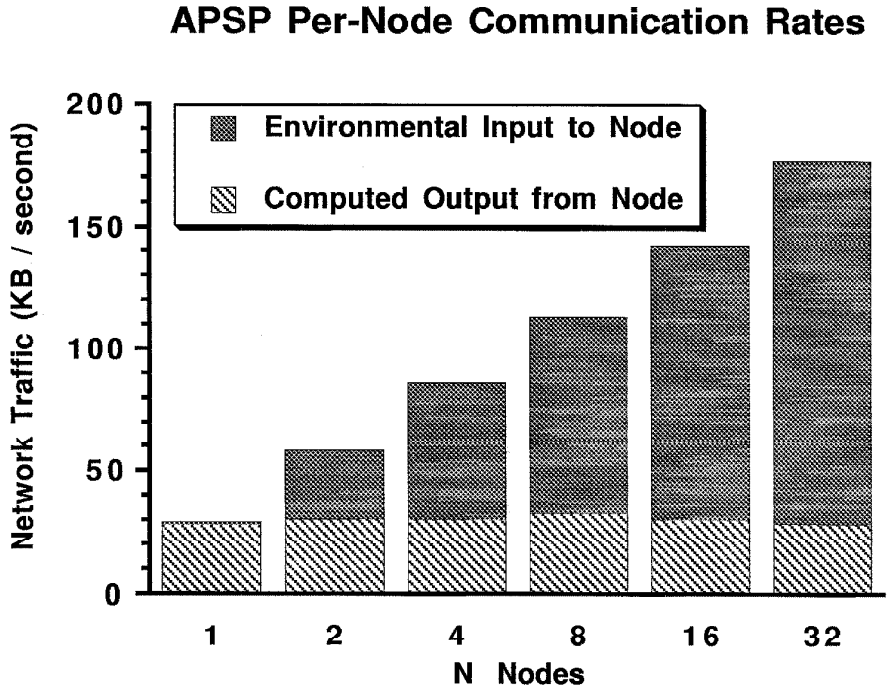

Figure 5.5: Per-node communication rates for All-Points Shortest Path computation. The amount of communication required to maintain the computational environment depends on both the logical connectivity of the problem and the way it is partitioned and mapped to the multicomputer nodes.

(see Section 5.3.1).

5.3.3 Per-Node Activity

Figure 5.4 shows total traffic more than doubling for each doubling of the number of nodes. Figure 5.5 examines the increasing per-node communication rates that cause this effect. The sketch of the APSP "vertex" action (Figure 3.30) helps explain the communication patterns of the program.

The "vertex" action finds a vector of minimum-cost paths by evaluating the cost of paths via each of its neighbors, and must access their cost vectors as well as its own. The action usually changes its output vector, causing it to be written back from the local node data-block cache to the node holding the master copy. That new output vector will be used as input by its eight neighboring actions, and must be propagated to the nodes where they reside.

As the Figure 5.5 demonstrates, an increase in the number of computational nodes increases the number of nodes to which updated versions of the data-block master copies must be sent. In the middle phase of the computation shown, each node generates

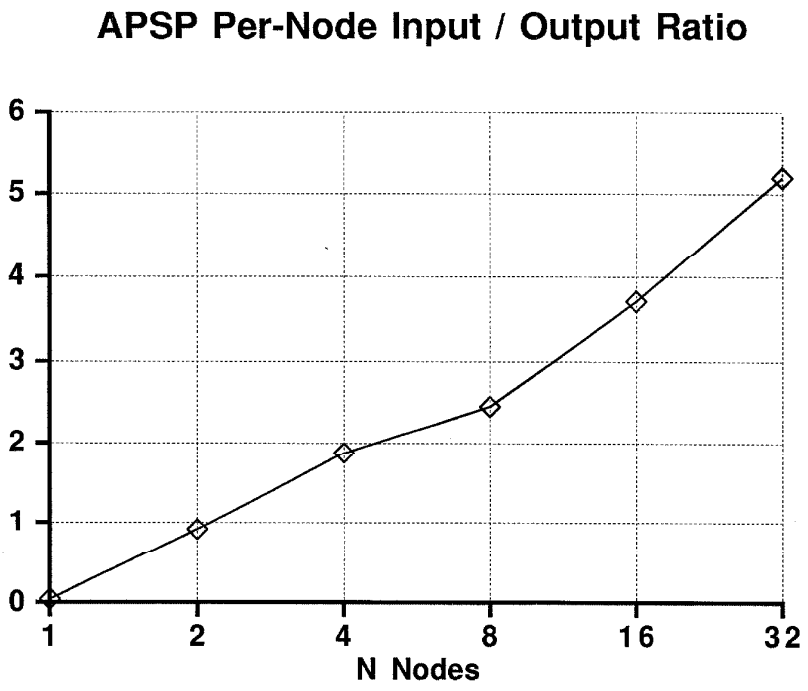

Figure 5.6: Ratio of input to output communication rates of Figure 5.5.

new and improved cost vectors on each activation, resulting in a nearly constant rate of updates (~110 Hz, see Figure 5.7) to the node containing the master copy for all configurations. Except for a small monitoring overhead, the one-node case receives no data from the master-service nodes because the node cache always contains the current version. In the two-node case, each cost vector will be needed by the other node, so the amount of input to the action node from the master servers is the same as the output of computed results to them. The "vertex" action computes a function in an environment of shared values. The cost of computing the function is relatively constant; the cost of maintaining the environment rises with concurrency.

The required degree of distribution of shared data depends on the logical connectivity of the problem, how it is partitioned, and the specific placement of actions. For example, a fully-connected graph would show distribution costs increasing linearly with the number of computational nodes. A good mapping of a large logical ring would have near-zero environmental update costs, a bad mapping would require each change to be sent to two other nodes. The sequential action placement chosen for this experiment causes a regular, recursive partitioning of the binary 8-cube with increasing N. For N = 4, new results must be distributed to just two other nodes, not all three. Figure 5.6 shows the ratio between the two components of Figure 5.5. The environmental overhead for this mapping of the 8-cube program, increases as the logarithm of N. The performance-monitor overhead tends to depress the ratio slightly below $log_2 N$.

The marginally higher value at N = 32 may reflect that the action-to-node mapping is non-ideal when the actions are placed on 27 nodes, breaking the power-of-two symmetry of the 8-cube partitioning.

5.4 Granularity and Ultimate Concurrency

As we have seen in Figure 5.3, the communication demands of even the larger configurations are not excessive. How far can we scale the concurrency of this problem? Without specific knowledge of the program's structure, an extrapolation of the rate of growth of the communications costs in Figure 5.3 would indicate that the program could be scaled to thousands of nodes. Of course, because of the medium-grain formulation of the program, the 8-cube program, as given, could not benefit from more than 256 nodes, putting one action on each multicomputer node. Greater concurrency requires either some amendment of the program or a larger problem.

Designing the APSP program with one vector and one action per vertex of the problem graph produces a simple and efficient program with adequate levels of concurrency for the S/2010 hardware. However, the individual elements of vector-minimum calculations are independent of each other, and the formulation is still correct if the vectors are split into independent fractions, each with its own "vertex" action. For example, halving the vector grain would give two actions per problem-graph vertex, each computing the paths from one-half of the graph vertices to itself. We observe that doubling or redoubling the number of actions in the program, in principle, need not change the total amount of data communicated between nodes, since the size of the vectors would decrease correspondingly. For this implementation, this variation is wasteful of space, because a fractional vector is smaller than the minimum data-block allocation.

Figure 5.7 shows the quickly diminishing returns from a fractional-vector experiment for the APSP 8-cube program. Each decrease in the grain size diminishes the number of vector operations over which the fixed scheduling and data-block-communications costs can be amortized. The communication costs of the full-size vector program grow logarithmically with increasing number of nodes (Figure 5.5) up to the formulation's natural limit at or near one action per node. Increasing the potential concurrency by using small vectors and more actions increases the total action costs linearly. The full-size 8-cube action performs a few thousand operations each activation. Figure 5.7 shows that the S/2010 implementation quickly becomes inefficient with smaller action granularity. However, these costs are characteristic of the implementation rather than the problem, and may be amenable to engineering solutions. The challenge of finer granularity is not meeting a boundless demand for network bandwidth, but efficient management and control of multitudinous small objects.

5.5 Latency and Blocking

In the previous section, we noted that our 8-cube APSP program could not effectively use more than 256 nodes. In fact, we would see significant drops in node-processor efficiency at that point, because we would no longer be able to successfully hide system communications latencies by overlapping parts of the execution cycles of multiple actions. Figure 5.3 shows that the node processor typically does not block on data-block

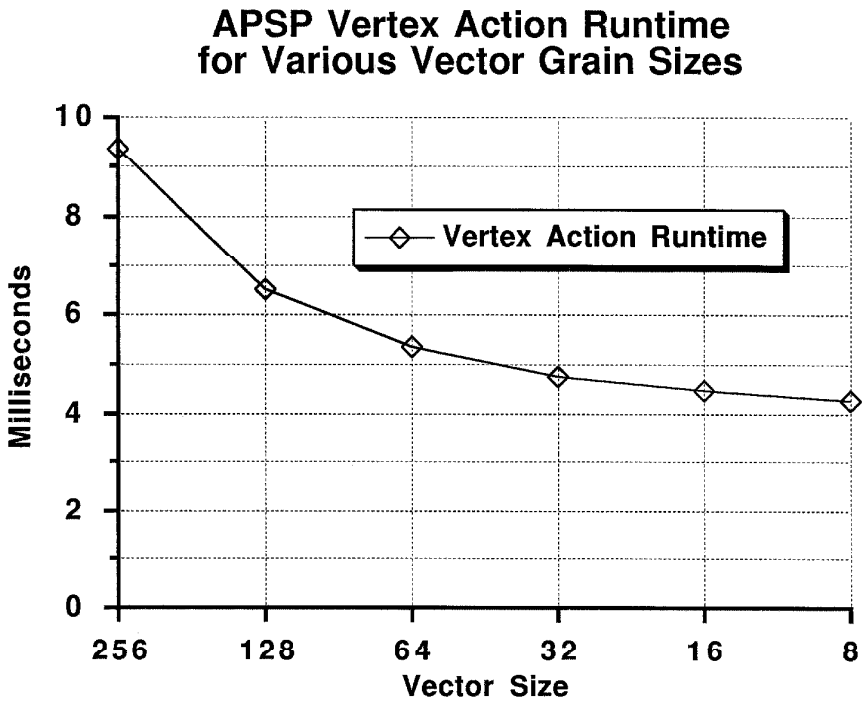

Figure 5.7: Runtime of APSP "vertex" action for various vector grain sizes, for 8-cube, N = 8. The APSP program runs correctly at smaller grain size, but action scheduling and communication overheads are constant, making small vectors inefficient.

access, nor idle waiting for action triggering. The optimistic data-block update policy and the inherent asynchrony of the Affinity model generally eliminate visible system latency. The kernel code scrupulously avoids RPC [10] operations, which would block while waiting for a reply, and overlaps communication latencies with action execution. Blocking is possible, but is rare except when a global resource is saturated. There are only two common circumstances causing node processors to block: startup, when all nodes are simultaneously fetching copies of newly-spawned action code and shared data structures, and printing to the host-console emulator, when many nodes may compete for that single low-bandwidth resource. The printer actions block as a form of flow control to avoid saturation and failure of the multicomputer-host workstation.

5.5.1 Action-Scheduling Costs

Figure 5.8 shows both the Affinity action scheduling cost and the degree to which latency is hidden. The "ring test" simply copies and decrements a counter from an input to an output data block. The actions are placed to ensure that logically connected actions do not reside on the same node except in the single-node case. In the N = 1 case, because the next action in the ring is on the same node, it can be triggered before the

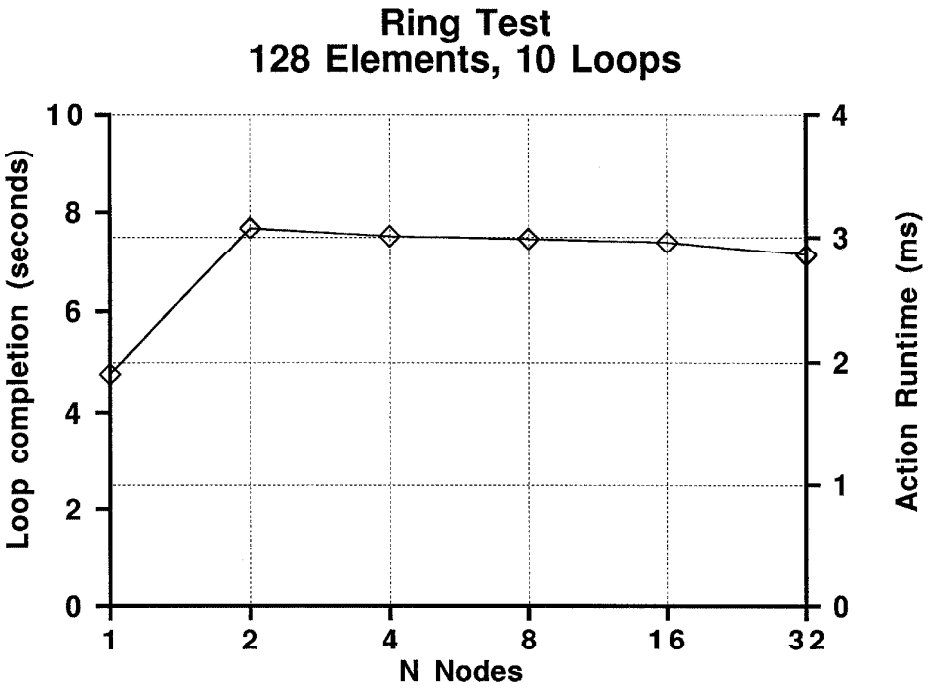

Figure 5.8: Action-scheduling costs. In the single node case, an optimization allows successful actions to trigger dependent actions before the master-copy update cycle is completed. The local cached copy of the data block is usable as soon as the success/failure decision is resolved but before the new version is sent to the node containing the master copy. In the multiple-node cases, a worst-case mapping exposes the latency of master-copy update.

data-block-update process is completed by the master-service node. The 1.1 millisecond difference between the one- and two-node cases is the real time required to send the new data block from one node to the master, have the master send an updated copy to a third node, and then have that node schedule an action in response to that triggering event.

5.6 Other Low-level Measurements

A few miscellaneous performance statistics for the S/2010 implementation are worth noting. The real time required to spawn a minimal action is about 9 milliseconds. Allocating a new minimum-sized data block takes 1.05 milliseconds, with an extra .09 milliseconds per additional page. The page-fault exception, which is used to record which data blocks are written by an action, costs 240 microseconds, with an extra 6 microseconds per additional page. A minimal system call, trapping to the CPU privileged mode, takes about 14 microseconds.

These times can be converted to instruction counts based on a nominal 4 MIPS rating of the 25 MHz Motorola 68020 node processor.

5.7 Fault Tolerance

The standard schemes for providing fault tolerance in concurrent systems require periodically making globally consistent checkpoints of the full system state. Typically, this may require halting the computation for several seconds [26]. Since Affinity does not maintain a globally synchronized system state in normal use, it does not require a global checkpointing scheme. Affinity has two distinct types of system state that must be preserved across faults: master copies of data blocks, and a record of the extant actions. Unlike process-based models, the transient state hidden within actions is inessential, reducing the amount of system state that must be saved. Tolerance of faults in the communication network requires comprehensive attention to the details of all message protocols; the problem is mitigated by the high reliability of multicomputer interconnects. Since the vast majority of the active electronic components are in the nodes rather than the network, we assume that node faults are of primary interest.

The issue of the master-copy recovery is not specific to Affinity or multicomputers, and can be implemented by database-transaction-logging techniques [9], which could be relatively fast if other nodes are used as the backing store. Since the action headers are themselves data blocks, the computation could be reconstituted from data blocks alone if we tag those containing action headers, but this would require a distinct kernel function to actively detect node failure, and a potentially widespread search to find all affected-action headers.

5.7.1 Single-Node Failures

If we assume that node failures are independent, we can restrict our attention to the problem of continuing through a sequence of single-node faults. We assume that the node will fault and stop quietly, without destructive network behavior, and without being able to send notification of its failure to other nodes. If the frequency of node failure is sufficiently low we can expect that we can usually perform the necessary recovery operations from a single-node fault-stop event before another node fault-stop complicates the issue.

5.7.2 Data-Block Master Copies

For master-copy recovery it is probably adequate to pair master-service nodes using a two-phase logging technique [9] to redundantly mirror the master copies. Since the pairing function can be determinate, any node detecting, by timeout, a possible master-service node failure can alert the mirroring secondary node. If there is a detected failure, the mirror node will notify all nodes with cached copies of its newly-primary data-block masters, and will establish a new mirroring relationship with yet another master-service node. The ability to track latent data-block references embedded within data structures (see Section 6.4.9) would allow all references to the defunct node to be purged neatly, but this is a nicety rather than a necessity.

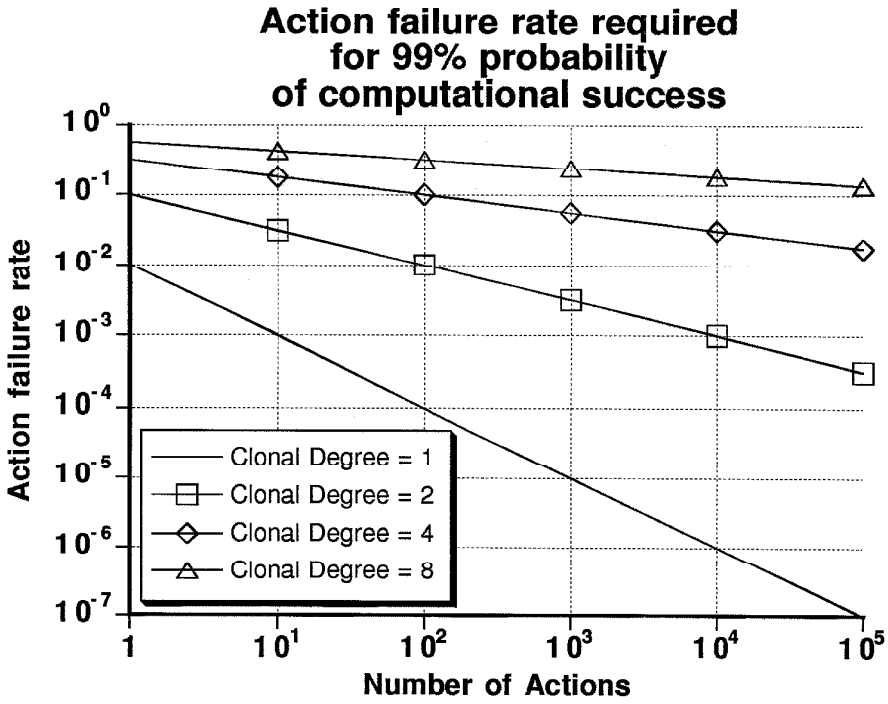

Figure 5.9: Required reliability of actions for high probability of computational success, assuming independent failure modes. The failure rate is that over the whole of the computation. Increasing degrees of redundancy by action cloning at the start of the computation raises the probability that at least one copy of each action will survive to the end. If we assume that the node failures are infrequent, we can dramatically improve reliability by recloning as needed during the computation to maintain redundancy lost due to node failures.

5.7.3 Action Cloning and Transparent Continuation

An action-executing node that detects a problem with a master-service node generally need not do anything except alert the mirror node to the difficulty; usually it can simply abort and reschedule the action. We know that action failures are always semantically correct provided that the action is rescheduled for future execution. This allows us to use action cloning (see Section 2.11.2) as a nearly-transparent mechanism for continuing a computation through single-node failures. As long as one clonal copy of an action is extant, the program will run correctly.

In the absence of some fault-tolerance mechanism, extraordinarily low node-failure rates are required for large-scale computations to complete successfully. Action cloning changes the scene, since the computation will succeed so long as one replica of each action survives to the end of the computation. Figure 5.9 shows that a computation with multiple action clones can tolerate much higher failure rates for a given probability

of success. This plot is based on the formula

$$F = (1 - .99^{1/N})^{1/K}$$

where F is the required action failure rate, N is the number of actions, and K is the cloning degree. Since the cost of cloning is at least linear in the degree, we would tend to favor low degrees of cloning. We note that two-fold redundancy yields the greatest benefit.

If failures are infrequent we can adopt a minimally-redundant two-clone policy and restore redundancy with a new clone after a lethal fault. It would not be too costly to have the action descriptor contain information about the number and frequency of execution of each clone. Such information would also be useful for deciding whether to increase or decrease the cloning degree for performance reasons as well. A significant disparity in the execution rate of clones of a particular action is an indication of node load inbalance, possibly in its most dramatic form, node failure. These statistics could be checked whenever an action is run, with further cloning if major disparities are evident. We must avoid placing clones on the same node, of course.

Action cloning may either improve or degrade effective concurrency, depending on the computation. If fault tolerance is our primary inducement to clone, a two-clone policy with replenishment would be adequately redundant against a sequence of infrequent single-node failures.

Figure 5.10 shows the performance effects of a two-clone policy on an APSP example. The program is computing the shortest-path matrix for a binary 7-cube, initially with four nodes executing actions. About two-thirds of the way through the most productive phase of the computation, one node fails and stops. The two two-clone lines, marked by hollow and black diamonds, initially are virtually identical, but diverge after the node failure. The rate of successful action execution drops quickly, not so much due to the direct loss of one-quarter of the hardware computational power as to the lack of triggering from actions on the defunct node. This experiment spawns the backup-clone actions at a lower priority to minimize write conflict failure rates. As the standard-priority actions begin to go idle due to lack of input events from the failed node, the backup actions become active. The initial activation of any action is slower than subsequent executions, so the post-failure rate drops to about the same level as at the startup. As the backup actions begin to run more efficiently after their initial execution, and trigger the intact primary actions, the action rate begins to climb again until cut short by the completion of the computation. The failing-node computation does tail off more slowly than the other cases due to poorer load balance after the failure. A longer-running computation would benefit from relocation of actions to rebalance the load, which is not done here. These tests used randomization in action placement and action scheduling within the node to reduce the possibility of scheduling artifacts.

The two-clone case with no failure also shows a small upward blip just before termination, also due to the lower priority of the backup clones, which run once, confirming without change the values computed by the standard-priority actions. The non-redundant single-action case runs at a high rate until its abrupt completion, since neither write conflicts nor lower-priority actions are involved.

Without a failing node, the two-clone case runs at almost the same peak rate as the non-redundant base case. It starts up more slowly, since twice as many actions are

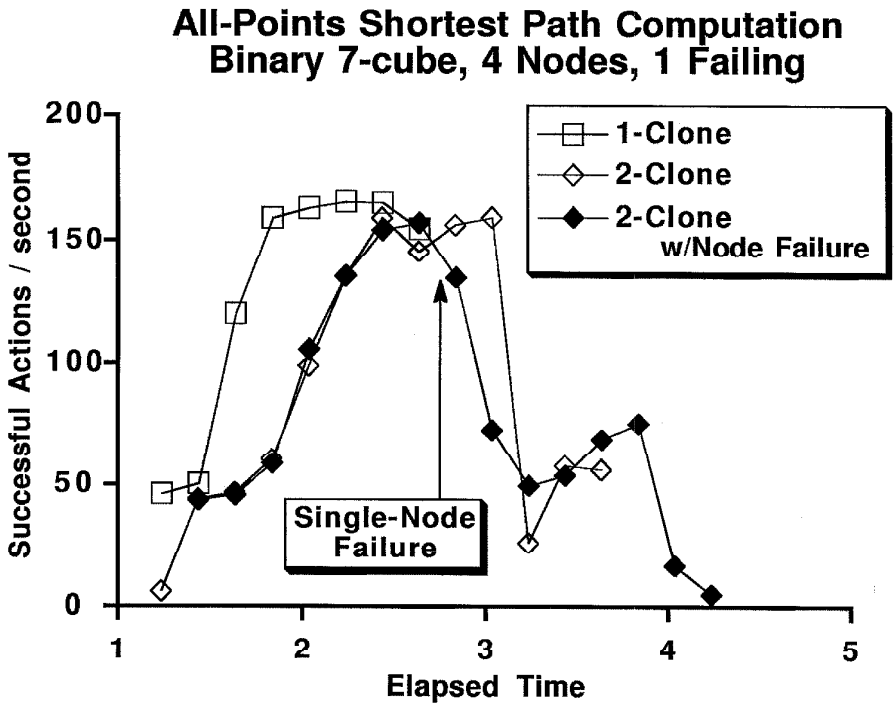

Figure 5.10: Effect of cloning and single-node failure on APSP productivity. The standard 1-clone case starts more quickly than the 2-clone cases because fewer actions are created. The actions are running on four nodes; the black-diamond line shows the effect of a single-node fault-stop in the middle of the computation. The line with single-node failure shows a significant drop in the rate of action execution and must run longer to complete the computation. No special detection or recovery mechanism is involved; the surviving-node kernels are not aware of the fault. The continuation of the computation through the fault is transparent, to the kernel as well as the user, except for the increased runtime.

spawned, and drags on a bit at the finish, to let the reserves into the game one time. In a long-running computation, these start- and end-point inefficiencies are minor compared to the ability to continue through a fault.

If we have redundant action clones and mirrored masters, we can expect that we will not require an active mechanism to determine node failure, which will instead be detected by another node in the normal course of program execution. Likewise, maintenance of action redundancy can be viewed as an aspect of a system load-balancing function.

5.7.4 Fault-Tolerance Summary

There are some hard problems glossed over in this discussion, such as ensuring a smooth changeover to a secondary master-service mirror if the primary node recovers rather than

stopping, and the problem of failure in the middle of a partially committed two-phase transaction. These errors are probably less likely than network failures; the sequential single-node fault-stop scenario is assumed to be the dominant failure mode.

While it is aesthetically attractive to avoid special-purpose fault-tolerance mechanisms and consider computational success in a probabilistic fashion, it is probably necessary to audit the system state at the end to confirm the result. A distinct validation operation may be required, but it should be easy to check for missing clones. Of course, data corruption can occur without causing a node fault-stop error. Multicomputer-hardware error-detection mechanisms are therefore important, but Affinity's inherent fault tolerance suggests that error-correction mechanisms may not be worthwhile, particularly if they may reduce system performance, e.g., error-correcting dRAM memory controllers.

Chapter 6

Summary and Evaluation

To assess the Affinity experiment, we pose and answer several questions:

What was the experiment?
First, developing an expressive computational model compatible with relaxed data coherence and highly distributed data and control (Chapters 2 and 3). Second, demonstrating that it was efficiently implementable on a medium-grain multicomputer (Chapters 4 and 5).

What are its successes and original contributions?
Sections 6.1 and 6.2 discuss these points at length, but, in summary, the combination of atomicity of effect, implicit write-set coherence, relaxed read-set coherence, and reactive scheduling produces a powerful and efficient programming system.

What are the interesting flaws and omissions?
Section 6.3 argues that some problems are more apparent than real and presents some programming techniques to circumvent potential pitfalls. Section 6.4 lists some omissions of the experimental implementation.

What is the significance for future concurrent systems?
Section 6.5 addresses this final question as best we can.

6.1 Recapitulation

The design decisions made for Affinity hang together well. The computational model is appropriate for multicomputers with capable communications systems, and is reasonably efficient and scalable. The implicit coherence mechanism provides good programming expressivity in comparison with other low-level systems. The externalization of computational state and scheduling provide additional flexibility in load balancing and fault tolerance.

6.1.1 Expressivity

Though expressivity is a subjective measure, the example programs are demonstrably concise. The APSP program is smaller than comparably efficient sequential code. The bounded-buffer code compares favorably in size to textbook examples using explicit

consistency mechanisms. The matrix multiply is virtually identical to a sequential implementation.

The Affinity programming model gives these relatively short programs capabilities absent in more conventional concurrent programming systems. The bounded-buffer code, unaltered, works correctly for multiple producers or consumers, without fear of deadlock. After the APSP program has computed the minimal path matrix, it will react to changes in graph costs by a localized and partial recalculation. Explicitly adding such features in a more conventional system would dramatically complicate the code complexity.

6.1.2 Tidiness of Implementation

The decision to make data-block sharing and mutually-exclusive writing part of the programming model simplifies many lower-level tasks. The core functions of the kernel are the creation, update, and cacheing of data blocks. These operations provide the services required by the user-level programming model, but surprisingly few extensions are needed to support kernel-level functions. Features such as on-demand fetching of the code to a multicomputer node and code sharing among the actions on a node require no special mechanisms.

Affinity actions have minimal *essential* (semantically significant) context; data blocks contain virtually all of the computational state that persists between action activations. An instance of an action is specified by little more than references to its argument-list binding, code, and trigger-set. Action relocation is a minor variation of action creation. A short message containing two references sent from one node to another is sufficient to allow an action to be loaded. Of course, large amounts of *transient* context may be required during actual execution of an action. Data blocks are mapped, fetched, and cached as needed while the action is active.

Affinity programs have demonstrated reasonable efficiency over configurations ranging from one to tens of multicomputer nodes. The overhead associated with concurrency has the pleasant observed scaling property that it increases with actual concurrency, with a diminishing rate of increase. A particularly gratifying observation is that a two-node concurrent formulation of a standard problem (e.g., matrix multiply or APSP) can run faster than the one-node sequential version, so that concurrency is at least slightly beneficial in even a minimally concurrent configuration.

6.1.3 Experimental Conclusion

The Affinity experiment met its goals. The Affinity programming style is relatively familiar: things tend to work as programmers expect. The cavalier approach to read-set coherence was motivated by a desire to find a programming model tolerant of data staleness, accepting it as an inevitable concomitant of asynchronous computation. Reactive scheduling, implemented by triggers, provides a usable mechanism for this goal. The stronger write-set coherence guarantees complement the reactivity and eliminate the nastiest surprises.

Compared to message-passing systems, the subsumption of many details of communication and control flow into the programming model reduces the attention to detail required of the programmer. The distributed-memory implementation promises improved

scalability and performance compared to shared-memory hardware. The decentralization of control effected by reactive scheduling allows programmers to construct programs by composition of autonomous objects without detailed knowledge of their inner workings. Defining actions with the semantic properties of atomic transactions provides great flexibility for the system to optimize resource use and adjust actual concurrency by action replication. The potential for transparent continuation of computations in the face of computational node failures is significant in its own right. The observation that redundant actions do not invariably degrade performance is a pleasant bonus.

6.2 Original Contributions

> But thai wil sai, Comparisons ar odius: in deed, as it fals out, thai ar too odious. — *c* **1573** G. Harvey [47]

Affinity successfully demonstrates some features that are widely regarded as difficult for a distributed and concurrent programming system. We shall excerpt from some recent publications to establish a basis for comparison.

6.2.1 Implicit User-level Atomic Actions

We cite Argus [34] as a leading example of the state of the art of distributed transaction languages.

> Argus is unique because it provides atomic actions within a programming language. ... Argus does not free the programmer from concern with details of concurrency. The programmer must think about deadlocks and starvation and implement the code to avoid them when possible. Often deadlocks are program errors, but this is not always true... User-defined atomic types are complicated to implement, but are not needed very often [34].

This characterization of atomic transactions as being essentially unusable by the average MIT programmer is dismaying. The claim of this thesis is that even well-done efforts like Argus are flawed by their half-way approach. Argus is a standard process model **with** atomic actions, adding further complexity to an already-intricate existing model; nothing is simplified. Affinity is a model **of** atomic actions, stripped of most of the attributes of conventional processes. Affinity shares this distinction with UNITY, but has resolved some important issues to permit feasible multicomputer implementation, e.g., the use of triggers to allow an efficient and flexible scheduling mechanism. Compared with conventional process models, the relatively finer grain of the actions and the reactive scheduling mechanism of triggers simplifies reasoning about code. The intrinsic flexibility of the model permits the system considerable freedom to enhance concurrency and/or fault tolerance.

6.2.2 Tractable Relaxed Data-Object Coherence

Affinity has a good match between the granularity of the action and the granularity of the data block. The trigger-based scheduler links the two and provides a basis for relaxing the data-coherence model. We cite Bryant [13] for a summary of efforts to use relaxed coherence models for distributed shared memory:

> The weak consistency model does not require that processors see a consistent view of storage at the completion of each store operation. Instead, the model only requires storage to be consistent after synchronization points. The argument for weak consistency is based on the assumption that correct parallel programs do not access shared memory in an undisciplined manner, but instead use synchronization primitives to ensure that a consistent application-level view of the shared storage is maintained at all times. ... Programs that do not maintain this discipline are, by definition, faulty. (For the purposes of this paper we ignore such applications as chaotic relaxation that execute properly in spite of concurrent read and write accesses to shared storage.)

This statement refers to both distributed shared memory and true shared-memory multiprocessor schemes such as the Dash [31] release-consistency model. Affinity demonstrates that there is a usable middle ground between expensive strict-coherence models and chaos.

Distributed-shared-memory models struggle to find a suitable policy to manage an expensive and unwieldy mechanism. Without an organizing principle, the results are arcane, awkward, error-prone and unpredictably inefficient. Affinity does provide a kind of distributed shared memory, but, unlike other efforts, settles on a single coherence policy that is unique in being relaxed, general, and implicit. These attributes are important not only to the ease with which code may be written, but also to ensure that users may use data types from libraries without knowledge about their inner workings. The Affinity atomic-action model is at least as efficient and scalable as other DSM coherence policies, and minimizes problems with latency and deadlock.

6.2.3 Write-set Coherence

Most DSM systems consist of low-level services lacking any object orientation to structure access and coherence issues. Even a relatively sophisticated object-oriented approach such as Orca [7], which shares Affinity's approach to letting the user define one form (data block internal) of consistency by making objects coherent units, balks at operations on multiple objects.

> Our model does not support indivisible operations on a collection of objects. Operations on multiple objects require a distributed locking protocol, which is complicated to implement efficiently. Moreover, this generality is seldom needed by parallel applications. ... However, the model is sufficiently powerful to allow users to construct locks for multioperation sequences on different objects so arbitrary actions can be performed indivisibly.

We doubt the user will easily manage what the system designer has shunned as too difficult or costly. We also differ with the suggestion that atomic multi-object operations are rare in concurrent programs. They are the rule in any program that maintains any form of distributed database, and are convenient, if not absolutely required, in many Affinity programs. The Affinity write-set coherence model and its implementation make such operations easy and usually cheap.

6.2.4 Implicit Coherent Composition

It is a commonplace that it is desirable to structure a computation as a composition of modular components with well-defined interfaces; this is the *raison d'être* for C++. In Affinity, reactive triggering replaces large-scale control flow and removes scheduling and update as a concern for the programmer, easing the use of modular components. We can define a C++ class that allocates an interface data block to be used by the rest of the program's actions. The class constructor instantiates other data blocks and reactive actions as needed.

The `checkTermination` object used in Sections 3.8 and 3.9 is an example of such an autonomous reactive object. After the interface object is instantiated by the line:

```
checkTermination      *ct = new checkTermination;
```

a trigger is set on it, and a status method (`done()`) is checked to verify termination of the calculation. Its internal workings are taken for granted.

The implicit nature of the Affinity coherence rules makes object-oriented composition a convenient and natural way to manage code complexity. Libraries of autonomous reactive objects can be used by programmers without analyzing their internal coherence properties. Composition of Affinity code does not weaken the write-set coherence properties that components may rely upon; additional operations may, in fact, expand the write set and the coherent sphere. The explicit mechanisms of other approaches make libraries problematic; the issue of nested locks and transactions [34] is an open research topic.

6.2.5 Efficient Data Update

DSM implementations are prone to unpredictable dramatic degradations of efficiency [13, 8, 32]. Affinity can also experience poor performance, but the modest size of action code makes it easier to anticipate and avoid. While it is true that some of the efficiency of the data-block sharing mechanisms are due to superior hardware features, e.g., a high-bandwidth network and small page sizes, the limiting resource in practice is the speed of a conventional processor.

Affinity's efficiency derives from the whole of the model, not from any trick of the implementation. Changing the transaction granularity from the variable to the object level is very significant. The atomic-action model gives code execution a comparable and consonant granularity. Weak read-set coherence and action atomicity allow optimistic execution and an efficient write-update policy [37]. Latency becomes virtually a non-issue for the medium-grain system of the S/2010 implementation, contrary to common DSM experience [37]. No single aspect of the Affinity model ensures efficiency, but some care was taken to ensure that no aspect requires inefficiency.

6.3 Difficulties and Complications

Affinity does have some quirks and weaknesses, notably the possible loss of concurrency by careless writing to shared variables. This section illustrates some ways to work around this problem. From the programmer's viewpoint, read-set incoherence is probably the most peculiar aspect of the Affinity model. Section 6.3.3 gives an example of how this

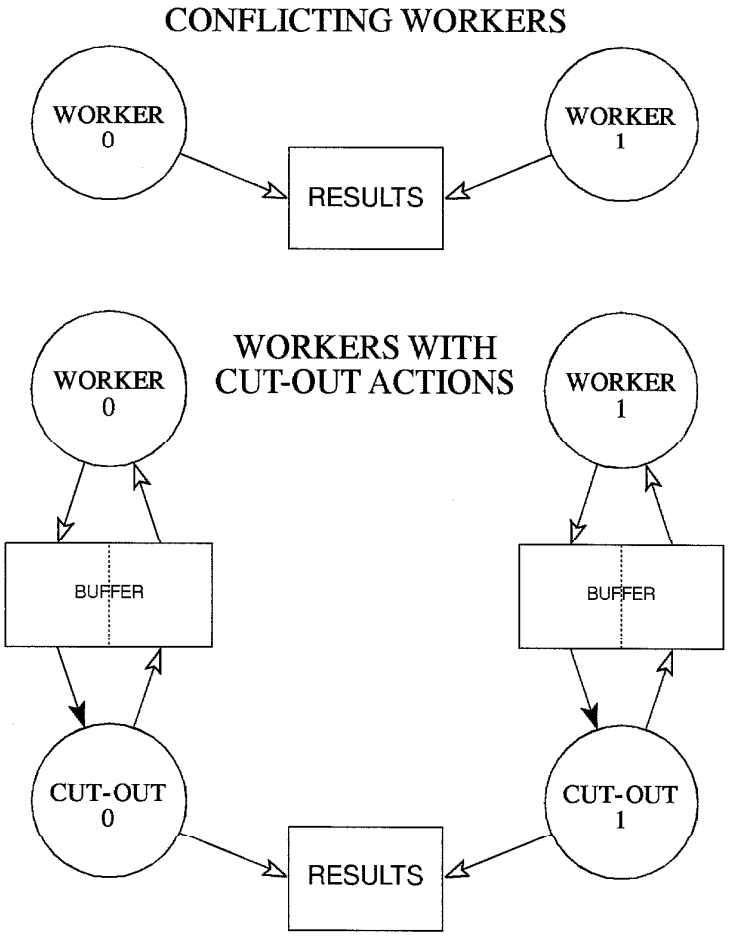

Figure 6.1: Top part of diagram shows serialization of actions due to write access conflicts. Lower part shows elimination of write access conflicts by the introduction of "cut-out" actions. "Cut-out" actions are expendible couriers of results.

novelty can be manifest in code. A deficiency of the implementation that detracts from compositional modularity is discussed in Section 6.4.2, the fact that quiescence is only globally detected.

6.3.1 Avoiding Unintended Serialization

An Affinity computation may be inadvertently serialized due to write conflicts to shared data blocks. The upper portion of Figure 6.1 shows schematically a write conflict that will serialize actions. Implicit mutual exclusion will allow only one "worker" action to succeed in modifying the "results" data block. Essentially, if there is any overlap in the execution times, only the first action to finish will succeed.

Buffered Cut-out Actions

The lower portion of Figure 6.1 shows one way of avoiding unintended action serialization. The "worker" actions do some significant amount of computation, then write the resulting data to a `buffer` object (as defined in Section 3.6). That buffer implementation has no write conflicts for a producer/consumer pair, and is therefore failure-free. Inserting the result in the buffer causes a "cut-out" [30] action to be scheduled. The cut-out action does nothing other than copying the computed result(s) to the "results" object. It may fail due to a write conflict, but since its execution time is minimal, it is both less likely to overlap executions with another "cut-out" action (reducing the probability of failure) and relatively "expendible" if it should fail. It will be scheduled repeatedly until it succeeds.

Spawned Cut-out Actions

Another variant of the "cut-out" scheme (not shown) is to have the "worker" action spawn a new "cut-out" action each time it produces a result. The newly-spawned action carries a result and the intended destination for that result in its argument list. The action code consists of a single assignment statement. Because it sets no triggers, it is a "one-shot" action, scheduled until it succeeds one time.

6.3.2 Task Queues

The more complex examples of Chapter 3, the matrix multiply and APSP programs, had their basic actions coded in a style that established a well-defined relation between input and output objects. Typically one action is associated with each data block of the computed result. This relational reactive style has many aesthetic and practical virtues, but may not be efficient or feasible for problems with large numbers of small tasks.

An alternative to this relational reactive action style is a task-queue scheme, a common idiom in a variety of concurrent-programming environments. Some number of worker actions carry out tasks provided from some centralized source, ultimately combining the results in a common destination. The program's logical concurrency is determined by the number of workers.

Task Dispatch

In Affinity the distribution of work from a single task queue is potentially serialized. Even though a task buffer is conflict-free for a single producer/consumer pair, there will be conflict between multiple consumers. This conflict can cause complete serialization of workers reading directly from a single buffer.

Figure 6.2 shows one possible solution to the task-dispatch problem. Tasks are removed from a centralized task queue by a single "dispatch & accrete" action, which is the sole writer to multiple per-worker input queues. The worker actions are triggered whenever new tasks are placed in their queues. The structural similarity of task dispatch and result accretion allows this program to fold the dispatch and accretion structures together, using the symmetric buffers introduced in Section 3.7. Depending on the details of the "dispatch & accrete" action code, the pipeline could be either supply- or demand-driven at any stage.

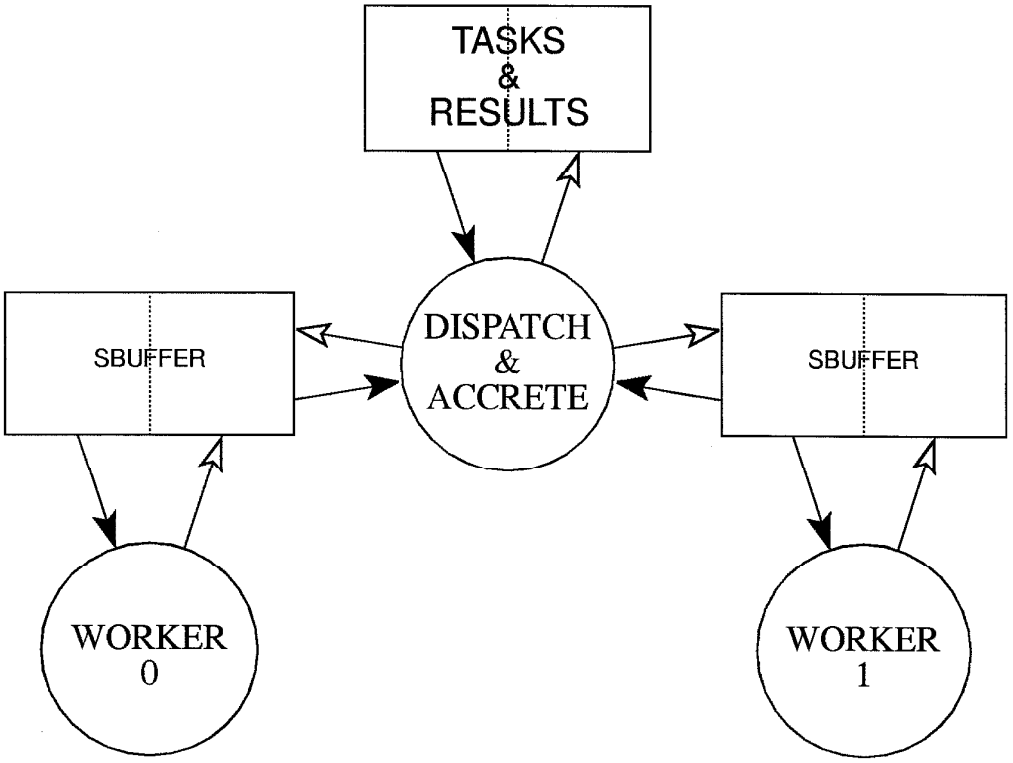

Figure 6.2: Folded task pipeline. Symmetric bidirectional bounded buffers are used to queue tasks and results at each stage. The folded pipeline is failure-free. Symmetry of triggering allows actions to be supply- or demand-driven.

6.3.3 Version and Message Disorders

Some appreciation of the implementation issues is helpful in seeing why the Affinity computational model says so little about read sets. Action cloning and, to a lesser extent, relocation, challenge an assumption likely to be internalized by programmers: read-set version monotonicity. Can an action running at a later time see an earlier version of a read-set data block than it did in a previous activation? Certainly. Affinity doesn't and couldn't guarantee read-set monotonicity. This expectation is natural if we anthropomorphize the actions; we expect a stream of data blocks to be delivered to us in nice order. However, unlike programmers, Affinity actions have no fixed identity. Suppose an action, which reads data block "A," executes on node "1" and is then instantly relocated to node "2." Without making strong assumptions about the communications network, we can't assert that the version of "A" is the same on both nodes. While instantaneous relocation is slightly contrived, the situation is the same and is

quite realizable when actions are cloned. (As an aside, we should observe that "later" has the intuitive meaning in this discussion. Because write-set variables are current, we can meaningfully define temporal ordering on actions with overlapping write-sets.) In practice, it is common to be able to observe that the read-sets may be skewed by one version.

Just how bad is this news? We have already met and resolved the problem at least once. We revisit the producer/consumer problem of Section 3.6. The beauty of the `buffer` class used in this example is that the logical object was implemented as two different data blocks to eliminate write conflicts (see Figure 3.13). The beastliness is that the write index may apparently go backwards as viewed by the consumer action. When the consumer reads an entry, it updates the current version of the read index, so there's no problem there. However, the write index, in a different block, may not be current.

We used a `buffer` class method `notempty()` to check whether there is an entry in the buffer, defined as

```
int   notempty()   { int &read_index = *read_index_p;
                     return read_index < write_index; }
```

which compares the read and write pointers.

Now imagine that another consumer clone has already read an entry from the buffer that our consumer action doesn't yet see. If we see the current version of the read index and the previous version of the write index, our view is that an entry has been read before it arrived. This is a transient inconsistency that is handled properly by the method above: since the read index is ahead of the write index, `notempty()` is false and the consumer action simply exits. But suppose we had coded the last line as:

```
return read_index != write_index;
```

which would be acceptable code if the read-set were always current. If the write index should become stale, the consumer action would incorrectly view the buffer as non-empty and would read an already-read entry, causing a real inconsistency and program error.

As seen in this example, incoherence in general, and read-set nonmonotonicity in particular, can produce subtle errors. Monotonicity tends to be ingrained in our thinking in one form or another. Similar errors in reasoning can appear in message-passing programs which often contain implicit requirements for message-order preservation and fail when a triangular race condition is encountered. Such problems give asynchrony a bad name. While Affinity's internal consistency and write-set coherence rules ease some problems, it may be difficult for programmers to embrace the realization that the first duty of an Affinity action is to know when to do nothing.

On the positive side, while network packet-order preservation simplifies low-level, multi-packet, message reassembly, it isn't required elsewhere by the model or the implementation.

6.4 Future Work

It would be a poor project indeed that did not hold out some interesting opportunities for further work. Most of those listed below are related to implementation improvements,

but the first are visible to the programmer.

6.4.1 Reactive User Interface

Debugger

Most of the state of an Affinity computation is "externalized" in data-block contents, visible to the system. A debugger could examine the state of any data block, and observe state changes without any special kernel support. It would be easy to provide an explicit master-copy locking service for a debugger; this would allow "single-stepping" the concurrent computation by preventing modification of a block while the lock was set.

Performance Monitor

The kernel gathers considerable information about node utilization, action runtime and success rates, etc. Performance monitoring software for highly concurrent systems is a challenging attraction, since the visualization techniques demonstrated for moderately concurrent systems are unlikely to scale well.

Reactive Spreadsheet Interface

One interesting possibility for both a demonstration program and user-interface tool is a reactive, concurrent spreadsheet. Spreadsheets are programmed in a reactive and relational way. As an interface tool, one can imagine interactive use of a concurrent system where a parameter is changed by the spreadsheet user, causing concurrent recalculations by reactive triggering.

6.4.2 Grouped Termination

Kernel-supported termination detection is a powerful tool for demonstrating the validity of results in Affinity's reactive environment. One weakness in the current system is that termination detection is globally determined. It would be desirable to be able to check that some grouped subset of all actions had terminated. This facility would improve the utility of libraries of autonomous reactive objects, and provide a form of barrier synchronization for distinct program components. This extension would also allow multiuser programming on nodes, if desired.

6.4.3 Limited Preemptive Scheduling

While Affinity supports priority scheduling, it does not provide preemption of actions. Preemption would be of value in real-time and multi-user systems, as well as various sorts of server objects. While completely general preemption would be prohibitively costly in terms of storage requirements, limited preemption could be supported at some increased cost (Section 4.13.2). The main implementation problem is that access conflicts to entries in the node data-block cache would have to be detected and resolved. The resolution could be as simple as aborting one of the conflicting actions. The detection of access conflicts violating atomicity guarantees would require recording not only write-accessed but also read-accessed data blocks. The overhead of read-access faults (which

would slow all actions) is a deterrent to this scheme, but a cache of read-access reference patterns might finesse the dilemma.

6.4.4 Persistence

A persistent object mechanism would be much better suited to the Affinity programming style than a conventional file system interface, which is ill-matched to reactive scheduling.

6.4.5 Improved Fault Tolerance

Without any special effort, Affinity computations can show remarkable tolerance for node fail-stop failures. However, a genuinely fault-tolerant system requires kernel support for master-copy logging, recovery, and relocation, as well as comprehensive node-failure detection.

6.4.6 Resource-Use Optimization

In many cases, action failures due to write conflicts reflect inherent serialization of the expressed program. Failure rates could be reduced, and efficiency improved, by relocating actions with repeated discovered conflict to the same node. The inverse of this approach is to allow the operating system to increase actual concurrency by replication of actions to multiple nodes. The Affinity model allows this to be done freely without semantic effect, but realizing improved performance by selective replication is an interesting problem.

Although update of data-block caches does not seem to be a major cost for the cases examined, other circumstances could make it desirable to reduce the number of cached copies maintained by concentrating actions with common references in common nodes. This is a familiar but difficult optimization problem, which could be solved concurrently [50].

6.4.7 Lighter Weight Actions

Depending on the details of computational node processor architecture (e.g., number of address bits), it might be significantly advantageous to have but a single virtual-address context for all actions on a node. At the extreme, one could have a common address space for all nodes, which would eliminate the distinction between the native and portable pointer types, improving efficiency and eliminating a coding complication.

6.4.8 Finer-Grained Hardware Implementation

Processors without memory-management units could run Affinity at a somewhat higher cost by software detection of dirtied pages. A check every time a pointer was used might be rather costly; a clever compiler might be able to reduce this burden. Munin [8], Orca [7], and Bryant's work [13], which address the problem of poor hardware support for DSM, may be helpful.

6.4.9 Reference Tracking

The current Affinity implementation has demonstrated that propagation of references in action code can be tracked as they are assigned and copied. The ability for the kernel to know what references (portable pointers) are contained in an action parameter list or data block opens several possibilities of real interest and utility:

Reference Optimization - Affinity currently does not perform "localization" optimizations on portable pointers in data blocks when they are dereferenced. Depending on the details of use, this can cause a significant runtime penalty. If the references were completely known, various optimization strategies might improve performance and reduce the overhead of concurrent codes compared to sequential versions.

Master Copy Relocation - Currently, master copies are not relocatable after creation. While forwarding could be used for storage-load balancing, the knowledge of the location of outstanding latent references would also improve recovery from node failure.

Master-Copy Garbage Collection - There is currently no mechanism for deletion of data-block master copies except for an explicit user-code request. This approach is tolerable in a batch-oriented, single-user environment; introduction of persistence, multiple users, etc., would require a more selective and effective garbage-collection mechanism. Identification of the extant references is needed for this purpose.

6.5 Prospects for Future Concurrent Systems

Since the S/2010 computer was not commercially successful, the S/2010 implementation of Affinity is of little significance in itself. However, the basic design of Affinity is portable to other medium-grain multicomputers. Two trends in current microprocessor design are troublesome for an Affinity implementation: relatively poor exception-processing performance and ever-larger address-translation-unit page sizes. The commercial impetus for high peak performance in workstation benchmarks would seem to cause a trend toward somewhat inflexible specialization in current processor designs. Nevertheless, the characteristics of current commercial multicomputers are fundamentally unchanged since the advent of the S/2010. Other implementations should perform adequately, subject to the caveat that expensive exception handling and large page sizes might necessitate larger granularity in programs.

The evolutionary path of increasingly large and fast workstation-like nodes is not the only one for multicomputer designs. There is a natural divergence between the ideal computational processor and a low-latency asynchronous-communications processor. Using a deeply-pipelined floating-point processor to handle interrupt-driven communications tasks is a poor use of silicon. The S/2010 implementation of Affinity is split between computational and communications tasks (see Section 4.3.3). An advanced multiprocessor design could probably benefit from similarly specialized node hardware. The Caltech Mosaic project [54, 5, 44] has produced an experimental fine-grain multicomputer based on a single-chip node. The dual-context processor and integral message-routing subsystem are very well suited for asynchronous communications

tasks. A direct port of Affinity to a Mosaic multicomputer is problematic due to the absence of an address-protection unit and the relatively small node dRAM size of 64KB. The first difficulty can be circumvented by software detection of data block access at somewhat higher cost. The restrictions on node memory require careful kernel design, but capable runtime systems can be built small [12]. The high performance of the communications network mitigates the per-node memory stringency, since the cost of communicating with remote node memories is low compared to medium-grain multicomputers. This is an active research area. We anticipate the development of a future multicomputer with hybrid characteristics well-matched to Affinity's requirements. The simplicity of the uninterpreted trigger mechanism allows even the task of action scheduling to devolve from the computational processor.

6.6 Conclusion

Grandly scaled multicomputers will require programming systems that, like Affinity, transparently replicate code and data, disperse control, and embrace asynchrony. Although no single aspect of Affinity is utterly novel, taken as a whole, the Affinity model is a root-and-branch excision of accreted debris that obscures the natural elegance of concurrent programming.

Bibliography

[1] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] R. Ananthanarayanan, M. Ahamad, R. J. LeBlanc, "Application Specific Coherence Control for High Performance Distributed Shared Memory," *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems* (SEDMS III), USENIX Assoc., Berkeley, CA, 1992.

[4] W. C. Athas, "Fine-Grain Concurrent Computations," Caltech Computer Science Technical Report TR:5242:87, 1987.

[5] W. C. Athas and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE COMPUTER*, August 1988.

[6] T. Axford, *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*, Wiley, Chichester, West Sussex, 1989.

[7] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, March 1992.

[8] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. 1990 Conf. Principles and Practice of Parallel Programming*, ACM Press, New York, NY, 1990.

[9] K. H. Bennett, "Mechanisms for Distributed Control," F. B. Chambers, ed., *Distributed Computing*, Academic Press, London, 1984.

[10] A. D. Birrel and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Vol. 2, No. 1, February 1984.

[11] N. J. Boden, "A Study of Fine-Grain Programming Using Cantor," Caltech Computer Science Technical Report CS-TR-88-11, 1988.

[12] N. J. Boden, "Runtime Systems for Fine-Grain Multicomputers," Caltech Computer Science Technical Report CS-TR-92-10, 1992, in preparation.

[13] R. Bryant, P. Carini, H-Y. Chang, B. Rosenburg, "Supporting Structured Shared Virtual Memory under Mach," *Proc. USENIX Mach Symposium*, USENIX Assoc., Berkeley, CA, 1991

[14] K. M. Chandy and J. Misra, *Parallel Program Design*, Addison-Wesley, Reading, MA, 1988.

[15] K. M. Chandy and S. Taylor, *Parallel Programming*, Jones & Barlett, Boston, MA, 1992.

[16] M. C. Chen, "Space-Time Algorithms: Semantics and Methodology," Caltech Computer Science Technical Report 5090:TR:83, 1983.

[17] E. W. Dijkstra,"Two Starvation Free Solutions to a General Exclusion Problem," EWD 625, Platannstraat 5, 5671 Al Nuenen, The Netherlands, 1978.

[18] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

[19] C. M. Flaig, "VLSI Mesh Routing Systems," Caltech Computer Science Technical Report 5241:TR:87, 1987.

[20] C. M. Flaig and C. L. Seitz, "Inter-Computer Message Routing System with each Computer having Separate Routing Automata for each Dimension of the Network," U. S. Patent 5,105,424, April 14, 1992.

[21] A. Forin, J. Barrera, M. Young, and R. Rashid, "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," Technical Report CMU-CS-88-165, Computer Science Department, Carnegie-Mellon University, August 1988.

[22] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[23] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Processors*, Vol. I: *General Techniques and Regular Problems*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[24] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[25] E. Jul, H. Levy, N. Hutchinson, A. Black, "Fine-Grained Mobility in the Emerald System," S. B. Zdonik and D. Maier, ed., *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, San Mateo, CA, 1990.

[26] M. F. Kaashoek, R. Michiels, H. E. Bal, A. S. Tanenbaum, "Transparent Fault-Tolerance in Parallel Orca Programs," *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems* (SEDMS III), USENIX Assoc., Berkeley, CA, 1992.

[27] V. E. Kotov, "On Parallel Languages," Chapter 5, J. Mikloško and V. E. Kotov, ed., *Algorithms, Software and Hardware of Parallel Computers*, Springer-Verlag, Berlin, 1984.

[28] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System," *Communications of the ACM*, Vol. 34, No. 10, March 1991.

[29] C. R. Lang, "The Extension of Object-Oriented Language to a Homogenous, Concurrent Architecture," Caltech Computer Science Technical Report 5014:TR:82, 1982.

[30] J. Le Carré, *Tinker, Tailor, Soldier, Spy*, Knopf, New York, NY, 1974.

[31] D. Lenoski, *et al.*, "The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos,CA, No. 2047, 1990.

[32] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Thesis, Department of Computer Science, Yale, 1986.

[33] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems*, Vol. 7, No. 4, November 1989.

[34] B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, Vol. 31, No. 3, March 1988.

[35] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft, *Operating Systems: Advanced Concepts*, Benjamin/Cummings, Menlo Park, CA, 1987.

[36] J. Y. Ngai, "A Framework for Adaptive Routing in Multicomputer Networks," Caltech Computer Science Technical Report CS-TR-89-09, 1989.

[37] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE COMPUTER*, August 1991.

[38] ParaSoft, "EXPRESS: a communication environment for parallel computers," Pasadena, CA, 1988.

[39] M. Pertel, "A Critique of Adaptive Routing," Caltech Computer Science Technical Report CS-TR-92-06, 1992.

[40] C. L. Seitz, "Concurrent VLSI Architectures," *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984.

[41] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol. 28, No. 1, January 1985.

[42] C. L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Seizović, C. S. Steele, W.-K. Su, "The Architecture and Programming of the Ametek Series 2010 Multicomputer," *Third Conference on Hypercube Concurrent Computers and Applications*, Vol. 1, pp. 33-36, ACM Press, 1988.

[43] C. L. Seitz, "Concurrent Architectures," Chapter 1, R. Suaya and G. Birtwistle, ed., *VLSI and Parallel Computation*, Morgan Kaufmann, San Mateo, CA, 1990.

[44] C. L. Seitz, "Multicomputers," Chapter 5, C. A. Hoare, ed., *Developments in Concurrency and Communication*, Addison-Wesley, Reading, MA, 1990.

[45] J. Seizović, "The Reactive Kernel," Caltech Computer Science Technical Report CS-TR-88-10, 1988.

[46] J. Seizović, "C+−," private communication, 1991.

[47] J. A. Simpson, *The Concise Oxford Dictionary of Proverbs*, Oxford University Press, Oxford, 1982.

[48] K. S. Smith and A. Chatterjee, "A C++ Environment for Distributed Application Execution," MCC Technical Report ACT-ESP-275-90, 1990.

[49] A. Z. Spector, "Distributed Transaction Processing and the Camelot System," Y. Paker, J-P. Banatre and M. Bozyiğit, ed., *Distributed Operating Systems: Theory and Practice*, Spring-Verlag, Berlin, 1987.

[50] C. S. Steele, "Placement of Communicating Processes on Multiprocessor Networks," Caltech Computer Science Technical Report 5184:TR:85, 1985.

[51] T. Stiemerling and T. Wilkinson, "Implementing DVSM on the TOPSY multicomputer," *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems* (SEDMS III), USENIX Assoc., Berkeley, CA, 1992.

[52] B. Stroustrup, *The C++ Programming Language*, second edition. Addison-Wesley, Reading, MA, 1991.

[53] W.-K. Su, "Reactive-Process Programming and Distributed Discrete Event-Simulation," Caltech Computer Science Technical Report CS-TR-89-11, 1989.

[54] "Submicron Systems Architecture Semiannual Technical Report," Caltech Computer Science Technical Report CS-TR-91-10, November 1991.

[55] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel," Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University, August 1987.

[56] F. Wieland, P. Reiher, D. Jefferson, "Experience in Parallel Performance Measurement: The Speedup Bias," *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems* (SEDMS III), USENIX Assoc., Berkeley, CA, 1992.